The dense multiple-vector tensor-vector product: An initial study

Nick Vannieuwenhoven *Nick Vanbaelen*
Karl Meerbergen *Raf Vandebril*

Report TW 635, September 2013

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

The dense multiple-vector tensor-vector product: An initial study

Nick Vannieuwenhoven *Nick Vanbaelen*
Karl Meerbergen *Raf Vandebril*

Report TW 635, September 2013

Department of Computer Science, K.U.Leuven

Abstract

The product of a dense tensor with a vector in every mode except one, called a tensor-vector product, is a key operation in several algorithms for computing the canonical tensor decomposition. In these applications, it is even more common to compute a tensor-vector product with the same tensor and r concurrently available sets of vectors, an operation we refer to as a multiple-vector tensor-vector product (MTVP). Current techniques for implementing these operations rely on explicitly reordering the elements of the tensor in order to leverage available matrix libraries. This approach has two significant disadvantages: reordering the data can be expensive if only a small number of concurrent sets of vectors is available in the MTVP, and this requires excessive amounts of additional memory. In this work, we consider two techniques resolving these issues. Successive contractions are proposed to eliminate explicit data reordering, while blocking tackles the excessive memory consumption. The numerical experiments on a wide variety of tensor shapes indicate the effectiveness of these optimizations, clearly illustrating that the additional memory consumption can be limited to tolerable amounts, generally without sacrificing expeditious execution. For several fourth-order tensors, the additional memory requirements were three orders of magnitude smaller than competing implementations, while throughputs of upward of 75% of the peak performance of the computer system can be attained for large values of r.

Keywords : tensor-vector product, multiple-vector tensor-vector product, canonical tensor decomposition, successive contractions, blocking, slicing.
MSC : Primary : 15A69, 65Y20, 65F30.

THE DENSE MULTIPLE-VECTOR TENSOR-VECTOR PRODUCT: AN INITIAL STUDY

N. VANNIEUWENHOVEN, N. VANBAELEN, K. MEERBERGEN, AND R. VANDEBRIL

Abstract. The product of a dense tensor with a vector in every mode except one, called a tensor-vector product, is a key operation in several algorithms for computing the canonical tensor decomposition. In these applications, it is even more common to compute a tensor-vector product with the same tensor and r concurrently available sets of vectors, an operation we refer to as a multiple-vector tensor-vector product (MTVP). Current techniques for implementing these operations rely on explicitly reordering the elements of the tensor in order to leverage available matrix libraries. This approach has two significant disadvantages: reordering the data can be expensive if only a small number of concurrent sets of vectors is available in the MTVP, and this requires excessive amounts of additional memory. In this work, we consider two techniques resolving these issues. Successive contractions are proposed to eliminate explicit data reordering, while blocking tackles the excessive memory consumption. The numerical experiments on a wide variety of tensor shapes indicate the effectiveness of these optimizations, clearly illustrating that the additional memory consumption can be limited to tolerable amounts, generally without sacrificing expeditious execution. For several fourth-order tensors, the additional memory requirements were three orders of magnitude smaller than competing implementations, while throughputs of upward of 75% of the peak performance of the computer system can be attained for large values of r.

1. Introduction

Multidimensional data arises naturally in several applications and is ubiquitous in engineering and sciences; we relate an illustrative example expounded in Lorente et al. (2010) that comes from aerospace engineering where numerical simulations of the airflow around an airfoil are a common and cost-effective procedure for design prototyping prior to expensive wind tunnel testing. In the simplest setting, such computational fluid dynamics simulations yield two-dimensional data: for every discretization point we have three spatial coordinates, three velocity components, the pressure and the temperature. These simulations are performed for various configurations of the airfoil corresponding to the regimes to which the foil may be subjected in the course of flight; this leads to additional dimensions corresponding to the angle of attack, yaw angle, the Reynolds number, and the Mach number, among others. Such simulations result in immense amounts of data, and there is a desire for compressing it without loosing information. In this context, tensor decompositions are a natural approach for dealing with this data, explicitly catering to its

Key words and phrases. tensor-vector product, multiple-vector tensor-vector product, canonical tensor decomposition, successive contractions, blocking, slicing.

†The first author acknowledges the support of a PhD Fellowship of the Research Foundation – Flanders (FWO). The third and fourth author acknowledge support by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office, Belgian Network DYSCO (Dynamical Systems, Control, and Optimization). The third author additionally acknowledges the support of KU Leuven Research Council grants PFV/10/002 (Optimization in Engineering) and OT/10/038 (Multi-parameter Model Order Reduction and its Applications), while the fourth author was additionally supported by the Research Foundation – Flanders (FWO) project G034212N (Reestablishing Smoothness for Matrix Manifold Optimization via Resolution of Singularities) and the KU Leuven Research Council project OT/11/055 (Spectral Properties of Perturbed Normal Matrices and their Applications).

multidimensional nature. The key to several tensor decompositions, such as the rank decomposition (Hitchcock, 1927) (also known as the Candecomp/Parafac (CP) decomposition (Carroll and Chang, 1970; Harshman, 1970)), block term decompositions (De Lathauwer, 2008), and orthogonal Tucker decompositions (Tucker, 1966; De Lathauwer et al., 2000a; Vannieuwenhoven et al., 2012) lies precisely in their ability to separate and relate information resulting from different dimensions, sometimes even admitting a meaningful interpretation.

Multidimensional data can be organized as a d-array of numbers

$$\mathcal{A} = [\![a_{i_1,i_2,\ldots,i_d}]\!]_{i_1,i_2,\ldots,i_d=1}^{n_1,n_2,\ldots,n_d} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d},$$

which may be considered as a coordinate representation of an abstract tensor $\mathcal{A}$ with respect to the standard tensor basis $\{\mathbf{e}_{i_1}\otimes\mathbf{e}_{i_2}\otimes\cdots\otimes\mathbf{e}_{i_d}\}_{i_1,i_2,\ldots,i_d=1}^{n_1,n_2,\ldots,n_d}$; herein, $\mathbf{e}_{i_k}$ is the i_kth standard basis vector of $\mathbb{R}^{n_k}$ and $\otimes$ denotes the tensor product. In this manner, the tensor $\mathcal{A}$ may formally be written as

$$\mathcal{A} = \sum_{i_1=1}^{n_1}\sum_{i_2=1}^{n_2}\cdots\sum_{i_d=1}^{n_d} a_{i_1,i_2,\ldots,i_d}\mathbf{e}_{i_1}\otimes\mathbf{e}_{i_2}\otimes\cdots\otimes\mathbf{e}_{i_d}. \tag{1}$$

We make no notational distinction in this text between the abstract tensor $\mathcal{A}$ that lives in $\mathbb{R}^{n_1}\otimes\cdots\otimes\mathbb{R}^{n_d}$, i.e., the tensor product of vector spaces, and its coordinate representation, the array $\mathcal{A}\in\mathbb{R}^{n_1\times\cdots\times n_d}$. We will sometimes refer to $\mathbb{R}^{n_i}$ as the ith *factor* of this tensor product of vector spaces and to n_i as the length or size of the ith factor. Tensors have an associated multilinear algebra, in which a multilinear transformation from $\mathcal{A}\in\mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$ to $\mathcal{B}\in\mathbb{R}^{m_1\times m_2\times\cdots\times m_d}$ via a set of matrices $\{M_k\in\mathbb{R}^{m_k\times n_k}\}_{k=1}^d$ is well-defined, and given by:

$$\mathcal{B} := \sum_{i_1=1}^{n_1}\sum_{i_2=1}^{n_2}\cdots\sum_{i_d=1}^{n_d} a_{i_1,i_2,\ldots,i_d}(M_1\mathbf{e}_{i_1})\otimes(M_2\mathbf{e}_{i_2})\otimes\cdots\otimes(M_d\mathbf{e}_{i_d}).$$

Following de Silva and Lim (2008), we write this operation as

$$\mathcal{B} = (M_1, M_2, \ldots, M_d)\cdot\mathcal{A} = (M_1^T, M_2^T, \ldots, M_d^T)^T\cdot\mathcal{A}.$$

Note that this operation is linear in every factor. The mode-k *tensor-vector product* (k-TVP) is a special kind of multilinear transformation, defined as:

$$\mathbf{v}_k \overset{k}{=} (\mathbf{v}_1,\ldots,\mathbf{v}_{k-1},I,\mathbf{v}_{k+1},\ldots,\mathbf{v}_d)^T\cdot\mathcal{A}, \tag{2}$$

where $\{\mathbf{v}_i\in\mathbb{R}^{n_i}\}_{i=1}^d$. The notation $\overset{k}{=}$ stresses that both sides of the equality should be interpreted as vectors in $\mathbb{R}^{n_k}$. Whenever a set of k-TVPs is performed with the same tensor but with different vectors, we refer to such an operation as a *mode-k multiple-vector tensor-vector product* (k-MTVP).

The k-TVP is a cornerstone of many algorithms for computing CP and orthogonal Tucker decompositions, which themselves are ubiquitous in applications; see, e.g., the overview articles (Kolda and Bader, 2009; Mørup, 2011). The k-TVP is the key operation in many algorithms for computing the best rank-1 approximation to a tensor: it is the core of alternating least-squares (ALS) algorithms such as those in (Kroonenberg and de Leeuw, 1980; De Lathauwer et al., 2000b; Kofidis and Regalia, 2002); it appears in the computation of the gradient required by most optimization-based algorithms, such as in (Zhang and Golub, 2001; Chang et al., 2010; Chen et al., 2012; Chen, 2012); and we should also mention the algorithm in (Kolda and Mayo, 2011) for computing eigenpairs of symmetric tensors, which can be considered a more general problem than computing only the best rank-1 approximation. Best rank-1 approximations may be leveraged in successive, or greedy, approximation algorithms for computing good rank-r approximations to a tensor; for instance, as in (Wang and Qi, 2007), and in the context of the proper generalized decomposition (PGD) as in (Ammar et al., 2010; Falcó and Nouy, 2012). The k-TVP is also a

key operation in some algorithms for computing an orthogonal Tucker decomposition, such as the methods considered in (Savas and Eldén, 2013; Goreinov et al., 2012).

The k-MTVP frequently appears in algorithms for computing a rank-r CP decomposition, even though the operation is often not introduced as such; it usually appears as Ref. M1, presented in section 3.1. The k-MTVP is typically the operation with dominant cost in the ALS algorithms in (Harshman, 1970; Carroll and Chang, 1970; Paatero, 1997; Phan et al., 2013a). The more complicated optimization algorithms from (Hayashi and Hayashi, 1982; Paatero, 1997; Tomasi and Bro, 2005; Oseledets and Savost'yanov, 2006; Acar et al., 2011; Phan et al., 2013b; Sorber et al., 2013a) require a k-MTVP (for every k) for constructing the gradient of the objective function. We mention in particular that MTVPs with $r \approx n$ are commonly occurring when computing CP decompositions. According to Bro and Andersson (1998) it is namely often computationally efficient to "compress" the tensor prior to computing a CP decomposition: given an $n_1 \times n_2 \times \cdots \times n_d$ tensor, one first computes the higher-order singular value decomposition (HOSVD) (Tucker, 1966; De Lathauwer et al., 2000b) using, for computational efficiency, the sequential truncation algorithm by Vannieuwenhoven et al. (2012). This results in a new basis for each factor, and a corresponding $\min\{n_1, r\} \times \min\{n_2, r\} \times \cdots \times \min\{n_d, r\}$ tensor—containing the coordinates with respect to this new basis—whereof the CP decomposition is subsequently computed.

Given the widespread use of tensors, it is unfortunate that general-purpose high-performance libraries for basic tensor operations similar to the basic linear algebra subroutines (BLAS) have yet to emerge, a concern also voiced by Schatz et al. (2013). This work aspires to contribute in this regard, investigating plausible performance-enhancing techniques for the (multiple-vector) tensor-vector product. It is the goal of this paper to compute the dense k-(M)TVP *efficiently*, both in terms of time and memory. In our opinion, a *generally applicable in-core* algorithm for tensor-vector multiplication should work within the memory restrictions imposed by the user without overly sacrificing execution time. We consider only *sequential* algorithms for *unstructured, dense* tensors fitting entirely in the main memory; this paper investigates neither techniques for structured (e.g., symmetric) tensors, sparse tensors, out-of-core algorithms, nor parallel implementations, although we do believe that the blocking techniques proposed in this paper can be useful in these contexts as well.

It appears that only recently the study of high-performance general-purpose algorithms for essential tensor operations, such as the TVP, MTVP and multilinear multiplication, has garnered interest from researchers. We should in particular mention the work by Schatz et al. (2013) who have recently systematically investigated blocking techniques for improving the performance of a symmetric multilinear multiplication with a symmetric tensor, and the idea of employing successive contractions for the MTVP in the work of Phan et al. (2013a).[1]

The prime contribution of this work is a time and memory efficient implementation of the k-MTVP relying on successive contractions and blocking techniques. The proposed final implementation, called Ref. M3B, will be shown to require two to four orders of magnitudes less additional memory than the currently widely dissipated approach Ref. M1, while concurrently consistently outperforming the latter in terms of execution time by a few percentage points.

The remainder of this paper is structured as follows. In section 2, three reference implementations of the k-TVP are presented and analyzed theoretically with respect to their time and space complexity. Section 3 deals with the k-MTVP and considers three reference implementations.

[1]The latter work deals with optimizing the computation of gradients of algorithms for computing CP decompositions. Hence it includes some optimizations that are not feasible in the context of the MTVP, and it is not clear from their numerical experiments that a large part of the realized performance gains are due to the successive contractions rather than because of storing the intermediary tensors, as we may deduce from our experiments.

Numerical experiments evaluating the reference implementations and the proposed optimizations are presented in section 4. Finally, our conclusions are presented in section 5.

Notation. In this document, vectors are typeset in a bold face font ($\mathbf{v}$), matrices as upper case letters (M, V), and tensors are typeset in a calligraphic font ($\mathcal{A}$, $\mathcal{B}$). Henceforth $\otimes$ denotes the Kronecker product and $\odot$ the Khatri-Rao product: $A \otimes B := \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \cdots & \mathbf{a}_n \otimes \mathbf{b}_n \end{bmatrix}$ where $\mathbf{a}_i$ and $\mathbf{b}_i$ are the columns of A and B respectively.

2. The single vector tensor-vector product

Given the wide applicability of the k-TVP, it is not surprising that several implementations of it exist; nevertheless, we only encountered software packages for Matlab. We begin by reviewing the implementation of this algorithm in three of these packages: the N-Way Toolbox v3.20 by Andersson and Bro (2000), the Tensor Toolbox v2.5 by Bader and Kolda (2006), and Tensorlab v1.0 by Sorber et al. (2013b).

The three investigated packages all employ the concept of *unfoldings* for computing the tensor-vector product. Unfoldings[2] were popularized in (De Lathauwer et al., 2000b; Bader and Kolda, 2006; Kolda and Bader, 2009) for efficiently implementing some tensor operations. The main idea is transforming tensor operations into familiar operations on matrices so as to take advantage of well-established and thoroughly optimized libraries implementing (parts of) the BLAS interface (Dongarra et al., 1988, 1990) such as ATLAS (Whaley and Petitet, 2005; Whaley et al., 2001), GotoBLAS (Goto and van de Geijn, 2008), Intel's MKL (Intel, 2013), Blaze (Iglberger et al., 2012), and Eigen3 (Guennebaud et al., 2010). The mode-k unfolding of $\mathcal{A}$, which is denoted by $\mathcal{A}_{(k)}$, is an $n_k \times \Pi_{i \neq k} n_i$ matrix whose columns are the mode-k vectors of $\mathcal{A}$; a mode-k vector $\mathbf{v}$ is a vector that is obtained by fixing all indices of $\mathcal{A}$, and varying only the index in factor k, i.e., $\mathbf{v} = \mathcal{A}_{i_1,\ldots,i_{k-1},:,i_{k+1},\ldots,i_d}$ with i_j a fixed value. The order of the mode-k vectors in the unfolding is determined by definition; we assume the canonical unfolding from Eldén and Savas (2009) in this paper.

The order in which the elements of the dense tensor are stored will influence the memory access pattern when computing unfoldings. In this paper, the standard approach encountered (implicitly) in the literature is assumed: the tensor is stored as the vectorization of the mode-1 unfolding. We will call this the *canonical vectorization*. With this convention, some unfoldings will require explicit reorganization of the data elements of the vectorized tensor, hereby increasing the memory consumption. Consider the following example of a tensor $\mathcal{A} \in \mathbb{R}^{4\times3\times2}$, where $a_{ijk} = 12(k-1) + 4(j-1) + i$, whose canonical vectorization is

$$\operatorname{vec}(\mathcal{A}) = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \end{bmatrix}^T,$$

and whose unfoldings are given by:

$$\mathcal{A}_{(1)} = \begin{bmatrix} 1 & 5 & 9 & 13 & 17 & 21 \\ 2 & 6 & 10 & 14 & 18 & 22 \\ 3 & 7 & 11 & 15 & 19 & 23 \\ 4 & 8 & 12 & 16 & 20 & 24 \end{bmatrix},$$

$$\mathcal{A}_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 13 & 14 & 15 & 16 \\ 5 & 6 & 7 & 8 & 17 & 18 & 19 & 20 \\ 9 & 10 & 11 & 12 & 21 & 22 & 23 & 24 \end{bmatrix},$$

$$\mathcal{A}_{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \end{bmatrix}.$$

[2] Also called "matricizations" and "flattenings."

By definition, the mode-1 unfolding requires no data reorganization if the tensor is stored with the canonical vectorization: one can simply interpret the linear array $\mathrm{vec}(\mathcal{A})$ as an $n_1 \times n_2 n_3$ matrix. With a typical implementation of the BLAS, both a matrix-vector and matrix product can be computed without allocating additional memory simply by providing the correct stride length to the appropriate routine. A similar observation holds for the mode-3 unfolding, which can be seen to be transpose of the matrix one obtains by interpreting $\mathrm{vec}(\mathcal{A})$ as a $n_1 n_2 \times n_3$ matrix. The mode-2 unfolding cannot be obtained without data permutations, hence requiring some additional memory for storing the unfolding if a 2-(M)TVP were computed. In general, explicit construction of the mode-1 and mode-d unfolding is not required for computing the corresponding (M)TVP; this will follow from (4) and (5).

In the complexity estimates of the considered implementations, we will include the number of operations required for computing the unfolding. That is because unfoldings typically represent a nonnegligible cost, firstly because of the integer computations for determining the correct offsets, and, secondly because of the irregular memory access pattern. We will count these operations as nonnegligible memory operations, or "mops." Memory operations in which index calculations are not necessary, e.g., for computing a mode-1 or mode-d unfolding, are not counted as their real cost is negligible. The complexity estimates do not reveal the relative costs of a floating-point and a memory operation, as the cost of the latter depends greatly on the type of memory that is accessed, i.e., register, caches, or main memory, and on the access pattern. It is important to stress that *the number of momps is thus not an estimate of the total number of data transfers from main memory to cache memory, but rather the number of costly data accesses attributable to the formation of unfoldings.* This convention will be used throughout.

2.1. **Reference S1: Unfoldings.** Using unfoldings, it is well-known (Kolda and Bader, 2009) that (2) is equivalent with

$$\mathbf{v}_k \overset{k}{=} \mathcal{A}_{(k)}(\mathbf{v}_1 \otimes \cdots \otimes \mathbf{v}_{k-1} \otimes \mathbf{v}_{k+1} \otimes \cdots \otimes \mathbf{v}_d). \tag{Ref. S1}$$

This leads to a first practical implementation founded on the dense matrix-vector product. We refer to it as Ref. S1. It is, essentially, used by the N-way Toolbox and Tensorlab in their optimization algorithms for the CP decomposition. In the Tensor Toolbox, an implementation of this strategy is offered by the `mttkrp` function.

Time and space complexity. The asymptotic time and space complexity depends on the order in which the expression tree for the binary Kronecker product operator is evaluated. In a typical scenario, we are not convinced that optimizing this evaluation order will significantly outperform the straightforward left-to-right evaluation. In this case, the number of operations is given by[3]

$$\mathrm{ops}_{\mathrm{S1}}(k, n_1, \ldots, n_d) = \mathcal{O}\left(2\prod_{j=1}^{d} n_j + \sum_{i=2}^{d-1} \prod_{\substack{j=1 \\ j<k}}^{i} n_j \prod_{j=k}^{i} n_{j+1}\right) \text{ flops } + \mathcal{O}\left(\prod_{j=1}^{d} n_j\right) \text{ mops.}$$

From this expression it can be seen that the computation of the matrix-vector product, which corresponds to the first term in the flop count, dominates the cost of computing the Kronecker product structured vector. The latter, with the left-to-right evaluation order, contributes at most $\frac{d-1}{2n_k}$ operations, relative to the former. Given that this fraction is usually much smaller than one, reducing it further with some optimized evaluation order will minimally affect the total execution time. We will therefore assume that the Kronecker structured vector in Ref. S1 is always computed using a left-to-right evaluation order. Remark that asymptotically only two floating-point computations are performed per memory access. One may therefore expect the performance of Ref. S1 to be memory-bound: the attainable throughput will be bounded by the

[3] "Flops" denotes "floating-point operations."

rate at which the elements of the unfolding are transferred into the cache memory, rather than by the number of floating-point operations that can be completed per clock cycle.

In general, the space complexity is $\mathcal{O}(2\prod_{j=1}^{d} n_j)$ values, because both the input tensor as well as its unfolding must be held in memory concurrently; however, if $k = 1$ or d then the complexity estimate halves due to the elimination of the explicit unfolding.

2.2. **Reference S2: Contractions.** In addition to an implementation of Ref. S1, the Tensor Toolbox by default pursues a different approach, springing from the observation that (2) may be written as

$$\mathbf{v}_k \overset{k}{=} (I, \ldots, I, \mathbf{v}_i, I, \ldots, I)^T \overset{d}{\underset{\substack{i=1\\ i\neq k}}{\cdot}} \mathcal{A} =: \mathbf{v}_i^T \overset{d}{\underset{\substack{i=1\\ i\neq k}}{\star}} \mathcal{A}, \tag{3}$$

where $\mathbf{v}_i$ is in the ith position. That is, the single k-TVP in (2) comprises $d-1$ multilinear multiplications that commute with one another, in which each such operation multiplies the tensor in just one factor, say i, with a vector, the result being a tensor whose ith factor is of length one; that is, it is essentially a tensor with a factor less. We will call a multilinear multiplication that applies a single vector in a single mode a *contraction* in this paper. The contractions in (3) can be processed in any desired order. Using unfoldings, the above can be computed sequentially with dense matrix-vector products. For notational simplicity, here we assume the contractions are effectuated from left-to-right: first the contraction with $\mathbf{v}_1$ in mode 1, then $\mathbf{v}_2$, and so on. These individual contractions may then be computed via unfoldings: letting $\mathcal{B}^0 := \mathcal{A}$,

$$\begin{aligned} &\mathcal{B}^1_{(1)} \leftarrow \mathbf{v}_1^T \mathcal{B}^0_{(1)}; && \mathcal{B}^2_{(2)} \leftarrow \mathbf{v}_2^T \mathcal{B}^1_{(2)}; && \cdots && \mathcal{B}^{k-1}_{(k-1)} \leftarrow \mathbf{v}_{k-1}^T \mathcal{B}^{k-2}_{(k-1)}; \\ \text{(Ref. S2)} \quad &\mathcal{B}^{k+1}_{(k+1)} \leftarrow \mathbf{v}_{k+1}^T \mathcal{B}^{k-1}_{(k+1)}; && \mathcal{B}^{k+2}_{(k+2)} \leftarrow \mathbf{v}_{k+2}^T \mathcal{B}^{k+1}_{(k+2)}; && \cdots && \mathcal{B}^{d}_{(d)} \leftarrow \mathbf{v}_{d}^T \mathcal{B}^{d-1}_{(d)}; \end{aligned}$$

where

$$\begin{cases} \mathcal{B}^j \in \mathbb{R}^{1\times\cdots\times 1\times n_{j+1}\times\cdots\times n_d} & \text{if } j < k \\ \mathcal{B}^j \in \mathbb{R}^{1\times\cdots\times 1\times n_k\times 1\times\cdots\times 1\times n_{j+1}\times\cdots\times n_d} & \text{if } j > k \end{cases},$$

so that $\mathbf{v}_k = \mathcal{B}^d_{(k)}$. We refer to this scheme as Ref. S2. In the Tensor Toolbox, this strategy is essentially implemented as the `ttv` function. Tensorlab can also compute the k-TVP in this manner by using the `tmprod` routine.

Time and space complexity. The number of operations may be verified to be

$$\mathrm{ops}_{\mathrm{S2}}(k, n_1, \ldots, n_d) = \mathcal{O}\left(2\sum_{i=1}^{k-1}\prod_{j=i}^{d} n_j + 2n_k \sum_{i=k}^{d-1}\prod_{j=i+1}^{d} n_j\right) \text{ flops} + \mathcal{O}\left(n_k \prod_{\substack{j=2\\ j\neq k}}^{d} n_j\right) \text{ mops}.$$

In this expression we only counted the dominant number of mops. It is equivalent with $\prod_{j=1}^{d} n_j$ if $k = 1$ and only $\prod_{j=2}^{d} n_j$ otherwise; that is because if $k = 1$, then a mode-2 unfolding must be computed explicitly, whereas the mode-1 unfolding requires no such computations. Regardless of this advantage, Ref. S2 is still memory-bound, asymptotically performing only two computations per memory access, because the number of data items that must accessed to compute the k-TVP is at least $\prod_{j=1}^{d} n_j$ regardless of k. Note that the dominant term in the number of flops amounts to $2\prod_{j=1}^{d} n_j$, so that Ref. S1 and Ref. S2 differ only in the lower-order terms. The order of the factors that minimizes the total number of floating-point operations for every k-TVP with that tensor may be understood to be $n_1 \geq n_2 \geq \ldots \geq n_d$.[4]

[4]Alternatively, but equivalently, the order in which the contractions are computed, which here we fixed to $\langle 1, \ldots, k-1, k+1, \ldots, d\rangle$, may be chosen to be a general permutation of the aforementioned sequence. However, for

The space complexity is $\mathcal{O}(\prod_{j=1}^{d} n_j + n_k \prod_{j=2, j\neq k}^{d} n_j)$ values, plus lower-order terms; the first term is the cost of storing the input tensor, the second term gives the number of values in the first unfolding that is explicitly computed.

2.3. **Reference S3: Successive contractions.** The main idea presented here appears to be understood partially by the numerical multilinear algebra community, but we were unable to locate a clear description of this important technique. None of the Matlab toolboxes implements this idea notwithstanding its simplicity and performance; hence, we are convinced that elaborating this technique is a worthwhile endeavor. From the implementation of the `ttv` routine in the Tensor Toolbox v2.5 it is obvious that its authors were aware of the right-to-left contraction detailed here. More recently, Phan et al. (2013a) considered alternative techniques to avoid expensive unfoldings, exploiting observations similar to the ones presented below. For computing one TVP, their "fast gradient from adjacent ones" is largely equivalent to our approach, except that we do not store the intermediate tensors because it is not applicable to the more general setting considered here. In the special case of computing gradients for CP decompositions, the approach in Phan et al. (2013a) may, at the cost of some additional memory, be expected to obtain a speedup of d over the approach presented here.[5]

As a k-TVP is memory-bound, one may expect that computing an unfolding contributes extensively to its execution time. The prime observation for reducing this cost is that permutations of the input data *can be avoided completely* for k-TVPs computed by Ref. S2. To show this, recall from (Eldén and Savas, 2009, Section 2.2) that

$$\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)} \in \mathbb{R}^{n_1 \cdots n_k \times n_{k+1} \cdots n_d} \quad \text{is defined by} \quad (\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)})_{I_1,I_2} := a_{i_1,\ldots,i_d}$$

where

$$I_1 := 1 + \sum_{l=1}^{k} (i_{k-l+1} - 1) \prod_{l'=1}^{l-1} n_{k-l'+1} \quad \text{and} \quad I_2 := 1 + \sum_{l=1}^{d-k} (i_{d-l+1} - 1) \prod_{l'=1}^{l-1} n_{d-l'+1}. \tag{4}$$

We assume here that an empty sum equals zero and an empty product equals one so that the above equations are well-defined for all $0 \leq k \leq d$. The case $k = d$ corresponds to the canonical vectorization of $\mathcal{A}$. From the above definitions, one can show that

$$\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)} = \mathcal{A}^{T}_{(k+1,\ldots,d;1,\ldots,k)}$$

by straightforward computations. Next, one discovers that an unfolding that does not require a reordering of the factors can be computed by reinterpreting linear memory as a matrix:

$$\operatorname{vec}(\mathcal{A}) := \mathcal{A}_{(1,\ldots,d;)} = \operatorname{vec}\left(\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)}\right), \tag{5}$$

for all $k = 0, \ldots, d$. That is, the vectorizations of the matrices $\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)}$ are equal, provided that $\mathcal{A}$ is stored in memory through the canonical vectorization. For proving (5), we know that

$$\left(\mathcal{A}_{(1,\ldots,d;)}\right)_I = a_{i_1,\ldots,i_d} \quad \text{with} \quad I = 1 + \sum_{l=1}^{d} (i_{d-l+1} - 1) \prod_{l'=1}^{l-1} n_{d-l'+1},$$

while, on the other hand,

$$\operatorname{vec}\left(\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)}\right)_{I'} = \left(\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)}\right)_{I_1,I_2} = a_{i_1,\ldots,i_d},$$

notational simplicity, we elected to present the results for a fixed processing order and to make recommendations with respect to the order of the factors of the tensor, such as we did here.

[5]We note that Phan et al. (2013a) do not investigate the performance gain originating from successive contractions *per se*, but rather investigate the total speedup resulting from both successive contractions and caching intermediary tensors.

provided that I_1 and I_2 are as in (4), and

$$I' := 1 + (I_2 - 1) + (I_1 - 1)N_2 \quad \text{with} \quad N_2 := n_{k+1} \cdots n_d.$$

By straightforward computations, i.e.,

$$\begin{aligned} I' &= 1 + \sum_{l=1}^{d-k} (i_{d-l+1} - 1) \prod_{l'=1}^{l-1} n_{d-l'+1} + \sum_{l=1}^{k} (i_{k-l+1} - 1) n_{k+1} \cdots n_d \prod_{l'=1}^{l-1} n_{k-l'+1} \\ &= 1 + \sum_{l=1}^{d-k} (i_{d-l+1} - 1) \prod_{l'=1}^{l-1} n_{d-l'+1} + \sum_{l=1}^{k} (i_{k-l+1} - 1) \prod_{l'=1}^{d-k+l-1} n_{d-l'+1} \\ &= 1 + \sum_{l=1}^{d-k} (i_{d-l+1} - 1) \prod_{l'=1}^{l-1} n_{d-l'+1} + \sum_{l=d-k+1}^{d} (i_{d-l+1} - 1) \prod_{l'=1}^{l-1} n_{d-l'+1} = I, \end{aligned}$$

the equality of the linearizations is obtained. Next, one notes that

$$\mathcal{A}_{(k)} = \mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)} \quad \text{if} \quad \mathcal{A} \in \mathbb{R}^{1 \times \cdots \times 1 \times n_k \times \cdots \times n_d}, \tag{6}$$

which follows from the definition of the vectorization of the right-hand side as before and from the definition of the canonical unfolding (Eldén and Savas, 2009):

$$(\mathcal{A}_{(k)})_{P,Q} = a_{i_1,\ldots,i_d} \quad \text{with} \quad P = i_k \quad \text{and} \quad Q = 1 + \sum_{\substack{m=1 \\ m \neq k}}^{d} (i_{d-m+1} - 1) \prod_{\substack{m'=1 \\ m' \neq k}}^{m-1} n_{d-m'+1}.$$

With the above definitions of P, Q, I_1 and I_2, one can also immediately show that

$$\mathcal{A}_{(k)}^T = \mathcal{A}_{(k+1,\ldots,d;1,\ldots,k)}^T = \mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)} \quad \text{if} \quad \mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_{k+1} \times 1 \times \cdots \times 1}, \tag{7}$$

by noting, after some computations, that $P = I_2$ and $Q = I_1$.

An efficient *left-to-right (LTR) contraction* that avoids computing unfoldings or permuting factors can be designed based on the above observations. Consider

$$\mathcal{B} := (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{k-1}, I, \ldots, I)^T \cdot \mathcal{A} = \mathbf{v}_i^T \overset{k-1}{\underset{i=1}{\star}} \mathcal{A},$$

where $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$, then, letting $\mathcal{B}^0 := \mathcal{A}$, we may compute,

$$\begin{aligned} \mathcal{B}^1_{(1;2,\ldots,d)} &\leftarrow \mathbf{v}_1^T \mathcal{B}^0_{(1;2,\ldots,d)}; \\ \mathcal{B}^2_{(1,2;3,\ldots,d)} &\leftarrow \mathbf{v}_2^T \mathcal{B}^1_{(1,2;3,\ldots,d)}; \\ &\vdots \\ \mathcal{B}_{(1,\ldots,k-1;k,\ldots,d)} &\leftarrow \mathbf{v}_{k-1}^T \mathcal{B}^{k-2}_{(1,\ldots,k-1;k,\ldots,d)}. \end{aligned} \tag{8}$$

Now, we note that in successive steps we need to transform the matrix $\mathcal{B}^i_{(1,\ldots,i;i+1,\ldots,d)}$ into $\mathcal{B}^i_{(1,\ldots,i+1;i+2,\ldots,d)}$, which involves *no computations at all*; by the above discussion, we can just reinterpret the linear memory vec $\left(\mathcal{B}^i\right)$ as an $n_{i+1} \times n_{i+2} \cdots n_d$ matrix. In a completely analogous fashion, the *right-to-left (RTL) contraction*

$$\mathcal{C} := (I, \ldots, I, \mathbf{v}_{k+1}, \mathbf{v}_{k+2}, \ldots, \mathbf{v}_d)^T \cdot \mathcal{A} = \mathbf{v}_i^T \overset{d}{\underset{i=k+1}{\star}} \mathcal{A},$$

where $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$, can be implemented efficiently as, letting $\mathcal{C}^0 := \mathcal{A}$,

$$\begin{aligned} \mathcal{C}^1_{(1,\ldots,d-1;d)} &\leftarrow \mathcal{C}^0_{(1,\ldots,d-1;d)} \mathbf{v}_d = (\mathbf{v}_d^T \mathcal{C}^0_{(d;1,\ldots,d-1)})^T; \\ \mathcal{C}^2_{(1,\ldots,d-2;d-1,d)} &\leftarrow \mathcal{C}^1_{(1,\ldots,d-2;d-1,d)} \mathbf{v}_{d-1}; \end{aligned} \tag{9}$$

$$\vdots$$

$$\mathcal{C}_{(1,\ldots,k;k+1,\ldots,d)} \leftarrow \mathcal{C}^{d-k-1}_{(1,\ldots,k;k+1,\ldots,d)}\mathbf{v}_{k+1}.$$

The successive unfoldings can again be accomplished through reinterpreting the linear memory. A k-TVP may then be computed as

$$\text{(Ref. S3)} \qquad \mathbf{v}_k = \mathbf{v}_i^T \overset{d}{\underset{\substack{i=1\\ i\neq k}}{\star}} \mathcal{A} = \mathbf{v}_i^T \overset{d}{\underset{i=k+1}{\star}} \left(\mathbf{v}_i^T \overset{k-1}{\underset{i=1}{\star}} \mathcal{A}\right)$$

by computing an LTR contraction followed by an RTL contraction. We refer to this implementation as Ref. S3. For completeness, a Matlab implementation is presented in appendix A.

Time and space complexity. The asymptotic time complexity is

$$\mathrm{ops}_{\mathrm{S3}}(k, n_1, \ldots, n_d) = \mathcal{O}\left(2\sum_{i=1}^{k-1}\prod_{j=i}^{d} n_j + 2n_k \sum_{i=k+1}^{d}\prod_{j=k+1}^{i} n_j\right) \text{ flops},$$

where the first sum is the number of operations attributable to the LTR contraction, and the second represents the operations in the RTL contraction. As all unfoldings are accomplished by reinterpreting linear memory, the number of mops is zero by definition. The dominant term in this estimate is still $2\prod_{j=1}^{d} n_j$, and the number of data transfers is roughly half of that, so that, asymptotically, only two operations are performed per memory access. This implementation is thus also memory-bound. Note that the ordering of the factors of the tensor minimizing the number of flops for a k-TVP is now $n_1 \geq \cdots \geq n_{k-1} \geq n_d \geq \cdots n_{k+1}$.

By eliminating explicit unfoldings, the space complexity becomes $\mathcal{O}((1+\frac{1}{n_1})\prod_{i=1}^{d} n_i)$ if $k \neq 1$, and $\mathcal{O}((1+\frac{1}{n_d})\prod_{i=1}^{d} n_i)$ otherwise, which asymptotically reduces the memory cost by 50%.

3. The multiple vector tensor-vector product

As with the performance difference between a matrix-vector product and a matrix-matrix product, it is not unexpected that the MTVP may be implemented more efficiently than simply repeating a number of TVPs. The issues and optimizations arising in this context are investigated in this section.

3.1. Reference M1: Unfoldings.

Consider the k-MTVP and let $\{\mathbf{v}_j^{(i)} \in \mathbb{R}^{n_j}\}_{i=1}^{r}$ for $1 \leq j \leq d$. We want to compute, for $i = 1, \ldots, r$,

$$\mathbf{v}_k^{(i)} \overset{k}{=} (\mathbf{v}_1^{(i)}, \ldots, \mathbf{v}_{k-1}^{(i)}, I, \mathbf{v}_{k+1}^{(i)}, \ldots, \mathbf{v}_d^{(i)})^T \cdot \mathcal{A}. \tag{10}$$

We can organize the vectors as matrices

$$V_j = \begin{bmatrix}\mathbf{v}_j^{(1)} & \cdots & \mathbf{v}_j^{(r)}\end{bmatrix} \in \mathbb{R}^{n_j \times r},\ j = 1, \ldots, d,$$

so that we immediately obtain from Ref. S1 and the definition of the Khatri-Rao product that

$$\text{(Ref. M1)} \qquad V_k \overset{k}{=} \mathcal{A}_{(k)}(V_1 \odot \cdots \odot V_{k-1} \odot V_{k+1} \odot \cdots \odot V_d),$$

where V_k is a matrix whose columns are the desired r vectors. This implementation of the k-MTVP, which is a matrix analogue of Ref. S1, is well-known, see, e.g., Kolda and Bader (2009), and it is employed in the three studied Matlab software packages. A computer implementation of Ref. M1 and Ref. S1 favors the former by a large margin since it is akin to the difference between r sequential applications of a BLAS2 matrix-vector product and a BLAS3 matrix multiplication. The latter admits higher throughput because its computational intensity, i.e., the average number of floating-point operations per data transfer, is higher.

We should remark that (10) is *not* equivalent with $(V_1, \dots V_{k-1}, I, V_{k+1}, \dots, V_d)^T \cdot \mathcal{A}$, i.e., the multilinear multiplication that results in a $r \times \cdots \times r \times n_k \times r \times \cdots \times r$ tensor instead of a matrix.

Time and space complexity. The time complexity of Ref. M1 is

$$\operatorname{ops}_{\text{M1}}(k, r, n_1, \dots, n_d) = \mathcal{O}\Bigg(2r \prod_{j=1}^{d} n_j + r \sum_{i=2}^{d-1} \prod_{\substack{j=1 \\ j \neq k}}^{i} n_j \prod_{j=k}^{i} n_{j+1}\Bigg) \text{ flops } + \mathcal{O}\Bigg(\prod_{j=1}^{d} n_j\Bigg) \text{ mops};$$

i.e., asymptotically the number of flops is r times that of $\operatorname{ops}_{\text{S1}}(k, n_1, \dots, n_d)$ while the number of mops remains unchanged. The computational intensity of the k-MTVP is now asymptotically $2r$ operations per data transfer, so that the throughput of a k-MTVP with r vectors may be expected to be significantly higher than computing r consecutive k-TVPs. For sufficiently large r this operation is namely compute-bound: the attainable throughput is limited by the number of floating-point operations that can be completed per clock cycle, rather than by the much slower rate at which the elements of an unfolding can be brought into the cache memory.

The space complexity is, in general, $\mathcal{O}((2 + \frac{r}{n_k}) \prod_{j=1}^{d} n_j)$; this represents the cost for storing the tensor, its unfolding, and the Khatri-Rao structured matrix. If $k = 1, d$, computing the unfolding is unnecessary, therefore reducing the coefficient to $1 + \frac{r}{n_k}$.

3.2. **Reference M3: Successive contraction.** A multiple vector analogue of Ref. S3 founded on matrix multiplication is also feasible.[6] Considering the LTR contraction (8) for each of the r vectors $\mathbf{v}_1^{(i)}$, it follows that the r matrix-vector products in the first step of (8) can be organized as a matrix multiplication:

$$\mathcal{B}'_{(1;2,\dots,d)} \leftarrow V_1^T \mathcal{B}^0_{(1;2,\dots,d)}. \tag{11}$$

After this operation, the rows of $\mathcal{B}'_{(1;2,\dots,d)}$ can be processed *individually* as in (8), continuing from the second step. To avoid reordering data, $\mathcal{B}'_{(1;2,\dots,d)}$ should be stored row-wise. Only the first step can be organized as a matrix multiplication, however, and all subsequent steps consist of matrix-vector products. In the case of a 1-MTVP, the above reasoning applies analogously for the first step of the RTL contraction in (9), allowing us to organize this step as a matrix multiplication.

The aforementioned scheme yields good performance in most cases, however, special care must be taken to handle a fringe case that does not arise with the TVP. The main problem is the following: computing a k-MTVP, $k \neq 1$, with r vectors and a $1 \times n_2 \times \cdots \times n_d$ tensor $\mathcal{B}$ would, by the previous observations, require a matrix product as in (11) where V_1^T is an $r \times 1$ matrix and $\mathcal{B}_{(1;2,\dots,d)}$ an $1 \times n_2 \cdots n_d$ matrix; that is, an *outer product of two vectors*, requiring $rn_2 \cdots n_d$ operations and as much temporary memory. This multiplies the memory consumption by a factor r when compared with sequentially computing r k-TVPs. The r matrix-vector products computing the contraction in the second mode require an additional $2rn_2 \cdots n_d$ operations. Ignoring the cost of the remaining operations, the total cost is proportional to $3rn_2 \cdots n_d$. On the other hand, if the tensor were organized as a $n_2 \times \cdots \times n_{d-1} \times 1 \times n_d$ tensor, then the number of operations would have been $2rn_2 \cdots n_d$ plus lower-order terms for computing the same k-MTVP. This ordering of the factors thus requires about one-third less operations.

Reordering the factors is not necessary to resolve the above issue, however. Let $\mathcal{A}$ be an $n_1 \times \cdots \times n_d$ tensor, $S = \langle i \mid n_i \neq 1 \rangle$ be an ordered set, $\ell = |S|$ the number of elements in this set, and let $\imath$ be the largest integer so that $s_\imath < k$ and let $\jmath$ be the smallest integer so that $k < s_\jmath$,

[6] The proposed scheme is similar to a technique considered in Phan et al. (2013a); however, caching of intermediary results is not appropriate in the present setting.

then[7]

$$\text{(Ref. M3)} \qquad \mathbf{v}_k^{(i)} \overset{k}{=} (\mathbf{v}_j^{(i)})^T \underset{\substack{j=1\\ j\neq k}}{\overset{d}{\star}} \mathcal{A} \overset{k}{=} \Big(\prod_{\substack{j\notin S\\ j\neq k}} \mathbf{v}_j^{(i)}\Big)\Big((\mathbf{v}_{s_j}^{(i)})^T \underset{j=\jmath}{\overset{\ell}{\star}} \Big((\mathbf{v}_{s_j}^{(i)})^T \underset{j=1}{\overset{\imath}{\star}} \mathcal{A}\Big)\Big),$$

where we exploited the observation that for $j \notin S$, $\mathbf{v}_j^{(i)} \in \mathbb{R}^1$ is just a scalar. Considering (6) and (8), and letting $\mathcal{B}^0 = \mathcal{A}$, the innermost contraction $\mathcal{B} := (\mathbf{v}_{s_j}^{(i)})^T \star_{j=1}^{\imath} \mathcal{A}$ can be computed by a modified LTR contraction:

$$\begin{aligned}
\mathcal{B}^1_{(1,\ldots,s_1;s_1+1,\ldots,d)} &\leftarrow (\mathbf{v}_{s_1}^{(i)})^T \mathcal{B}^0_{(1,\ldots,s_1;s_1+1,\ldots,d)};\\
\mathcal{B}^2_{(1,\ldots,s_2;s_2+1,\ldots,d)} &\leftarrow (\mathbf{v}_{s_2}^{(i)})^T \mathcal{B}^1_{(1,\ldots,s_2;s_2+1,\ldots,d)};\\
&\vdots\\
\mathcal{B}_{(1,\ldots,s_\imath;s_\imath+1,\ldots,d)} &\leftarrow (\mathbf{v}_{s_\imath}^{(i)})^T \mathcal{B}^{\imath-1}_{(1,\ldots,s_\imath;s_\imath+1,\ldots,d)}.
\end{aligned}$$

One may similarly derive the modified RTL contraction for the contraction with the vectors $\{\mathbf{v}_{s_j}^{(i)}\}_{j=\jmath}^{\ell}$ in (Ref. M3) from (7) and (9). Key here is the realization that this is still a successive contractions scheme, where the unfoldings required in successive steps can be obtained from reinterpreting the linear memory, because of the fact that $n_i = 1$ for every $s_j < i < s_{j+1}$ combined with (6) and (7). The first step in both the modified LTR and RTL contractions—that is, the step where the mode s_1, respectively s_ℓ, unfolding is multiplied with a vector—can be organized as a matrix multiplication as before, yielding an implementation of the k-MTVP that is suitable when some of the modes of the tensor have length 1. After performing the LTR contraction followed by the RTL contraction for computing the k-TVP on the right-hand side of Ref. M3, one should still multiply with the scalar $\prod_{j\notin S, j\neq k} \mathbf{v}_j^{(i)}$. We will refer to this scheme as Ref. M3.

A Matlab code implementing Ref. M3 is presented in appendix B.

Time and space complexity. The operation count for Ref. M3 is given by

$$\mathrm{ops}_{\mathrm{M3}}(k, r, n_1, \ldots, n_d) = \mathcal{O}\Big(2r\Big(\sum_{i=1}^{\imath}\prod_{j=i}^{\ell} n_{s_j} + n_k \sum_{i=\jmath}^{\ell}\prod_{j=\jmath}^{i} n_{s_j}\Big) + r(d-\ell+n_k)\Big) \text{ flops},$$

which is asymptotically r times $\mathrm{ops}_{\mathrm{S3}}(k, n_1, \ldots, n_d)$. The first summation is due to the modified LTR contraction successively contracting in the modes $s_1, s_2, \ldots, s_\imath$; the second summation represents the cost of the RTL contraction in the modes $s_\ell, s_{\ell-1}, \ldots, s_\jmath$; and the remaining term counts the number of operations required for computing the multiplication with the scalars $\prod_{j\notin S, j\neq k} \mathbf{v}_j^{(i)}$. The dominant term in the complexity estimate, i.e., $2r\prod_{j=1}^{d} n_j$, originates from the first matrix multiplication. This term equals the dominant term in the time complexity of Ref. M1. As the number of data transfers is also asymptotically equal to Ref. M1's, Ref. M3 is also compute-bound for large r.

The space complexity of Ref. M3 improves upon that of Ref. M1; as explicit unfoldings are avoided by the former, the complexity reduces to $\mathcal{O}((1+\frac{r}{n_{s_1}})\prod_{i=1}^{d} n_i)$ if $k \neq 1$, and $\mathcal{O}((1+\frac{r}{n_{s_\ell}})\prod_{i=1}^{d} n_i)$ otherwise.

A significant advantage of Ref. M3 over Ref. M1 concerns the aforementioned fringe case. From the latter's time complexity analysis it follows that if $n_k = 1$, then the number of operations is proportional to $3rn_1\cdots n_d$; two-thirds from the "matrix product", which actually is a vector-matrix multiplication, and another third attributable to constructing the Khatri-Rao structured

[7]By definition, $\jmath = \imath + 1$ if $s_{\imath+1} \neq k$, and $\jmath = \imath + 2$ otherwise.

matrix. From the analysis presented here, we see that Ref. M3 asymptotically requires about one-third less operations. An even greater disadvantage of Ref. M1 concerns its memory consumption in this fringe case: $rn_1 \cdots n_d$, which is r times the size of the input tensor, simply to create a $1 \times r$ vector.

Note that Ref. M1 stores asymptotically $(1 + \frac{r}{n_k})n_1 \cdots n_d$ values in addition to the input tensor if $k \neq 1, d$, whereas Ref. M3 only stores $\frac{r}{n_{s_1}} n_1 \cdots n_d$ additional values if $k \neq 1$. Note the difference in the denominator. With Ref. M1, if this fraction is large, the memory consumption cannot be reduced by choosing a different order of the factors of the tensor; in contrast, with Ref. M3, the factors can always be reordered prior to computing the k-MTVP so as to maximize n_{s_1}, hereby minimizing the additional memory requirements.

3.3. **Reference M3B: Successive contractions with blocking.** The prime performance bottleneck of the k-TVP is the cost of computing unfoldings, which was remedied by considering successive contractions yielding Ref. S3. In the case of the k-MTVP, it is clear from the complexity estimate of Ref. M1 that the relative cost of an unfolding rapidly declines as the number of vectors r increases, and, hence, becomes less of a liability with respect to Ref. M3. On the other hand, the memory cost may be understood to be increasing: considering an $n \times \cdots \times n$ tensor where $r \geq n$ the intermittent memory requirements are even larger than storing the input tensor, notwithstanding that the result of the operation is only an $n \times r$ matrix. Clearly this comprises a subpar performance for methods wishing to claim general applicability.

Here, blocking strategies are developed with the goal of performing an MTVP within the memory restrictions imposed by the user. We are inspired by high-throughput algorithms for matrix multiplication, such as (Goto and van de Geijn, 2008), which employ cache blocking techniques to attain high efficiency while not requiring additional main memory. Blocking for matrices has a long history and is consequently well-understood. However, for tensors it was only recently considered by Ragnarsson and Van Loan (2012). That work is chiefly concerned with the connection between blocked tensors and an unfolding resulting in a matrix with a natural block structure. While these developments might be used for proving the following results, we believe that a direct approach based on multilinear algebra is more natural and yields immediate algorithms. Furthermore, explicitly constructing unfoldings is often unnecessary, may impair performance, and increases memory consumption.

We begin our investigation by showing that the k-MTVP of a blocked tensor may be computed through multiple k-MTVPs with each of the subtensors. Let $\mathcal{A}$ be as in (1) and assume that we subdivide $\mathcal{A}$ into $q_1 \times \cdots \times q_d$ blocks of size $b_1 \times \cdots \times b_d$:

$$\mathcal{A} = \left[\mathcal{A}_{i_1,i_2,\ldots,i_d}\right]_{i_1,i_2,\ldots,i_d=1}^{q_1,q_2,\ldots,q_d}, \quad \text{where} \quad \mathcal{A}_{i_1,i_2,\ldots,i_d} \in \mathbb{R}^{b_1 \times b_2 \times \cdots \times b_d}, \tag{12}$$

i.e., $b_j q_j = n_j$, which we shall assume for the sake of brevity and unencumbered notation. Extensions to accommodate for fringe blocks and nonuniform blocking patterns are left as an exercise.[8] Consider now a matrix $S_{j_k}^T \in \mathbb{R}^{b_k \times n_k}$, $1 \leq j_k \leq q_k$, that "selects" the rows $(j_k - 1)b_k + 1$ through $j_k b_k$ if applied to a matrix with compatible dimensions; i.e.,

$$S_{j_k}^T = \begin{bmatrix} 0 & \cdots & 0 & I_{b_k} & 0 & \cdots & 0 \end{bmatrix},$$

where the $b_k \times b_k$ identity matrix is at position j_k, and each of the $q_k - 1$ zero matrices has dimension $b_k \times b_k$. Using these definitions, one finds:

$$(S_{j_1}, \ldots, S_{j_d})^T \cdot \mathcal{A} := (S_{j_1}, \ldots, S_{j_d})^T \cdot \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} (\mathbf{e}_{i_1}, \ldots, \mathbf{e}_{i_d}) \cdot a_{i_1,\ldots,i_d}$$

[8]The implementation we developed can handle these fringe blocks, but does not support more general nonuniform blocking patterns.

$$= \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} (S_{j_1}^T \mathbf{e}_{i_1}, \ldots, S_{j_d}^T \mathbf{e}_{i_d}) \cdot a_{i_1,\ldots,i_d}$$

$$= \sum_{i_1=(j_1-1)b_1+1}^{j_1 b_1} \cdots \sum_{i_d=(j_d-1)b_d+1}^{j_d b_d} (\mathbf{e}'_{i_1}, \ldots, \mathbf{e}'_{i_d}) \cdot a_{i_1,\ldots,i_d} = \mathcal{A}_{j_1,\ldots,j_d},$$

where $\mathbf{e}'_{i_k}$ is the i_kth standard basis vector in $\mathbb{R}^{b_k}$. In the above equations, the second equality is due to the multilinearity of the product, and the penultimate equality by straightforward computation. Thus, showing that a k-TVP with a blocked tensor is essentially a sum of k-TVPs with the subtensors becomes easy. Consider for the sake of brevity but without loss of generality, the 1-MTVP:

$$\begin{aligned}
\mathbf{v}_1^{(i)} &\overset{1}{=} (I, \mathbf{v}_2^{(i)}, \mathbf{v}_3^{(i)}, \ldots, \mathbf{v}_d^{(i)})^T \cdot \mathcal{A} \\
&\overset{1}{=} (I, \mathbf{v}_2^{(i)}, \mathbf{v}_3^{(i)}, \ldots, \mathbf{v}_d^{(i)})^T \cdot \Big(I, \sum_{j_2=1}^{q_2} S_{j_2} S_{j_2}^T, \ldots, \sum_{j_d=1}^{q_d} S_{j_d} S_{j_d}^T\Big) \cdot \mathcal{A} \\
&\overset{1}{=} \sum_{j_2=1}^{q_2} \cdots \sum_{j_d=1}^{q_d} (I, S_{j_2}^T \mathbf{v}_2^{(i)}, \ldots, S_{j_d}^T \mathbf{v}_d^{(i)})^T \cdot (I, S_{j_2}, \ldots, S_{j_d})^T \cdot \mathcal{A}, \qquad (13)
\end{aligned}$$

where in the second equality we note that the orthogonal projectors $S_{j_k} S_{j_k}^T$ sum to the identity matrix, and in the last equality the multilinearity property was exploited several times to move the sums out of the multiplication. Letting,

$$\mathbf{v}_k^{(i)} = \begin{bmatrix} \mathbf{v}_{k,1}^{(i)} \\ \vdots \\ \mathbf{v}_{k,q_k}^{(i)} \end{bmatrix} \quad \text{where} \quad \mathbf{v}_{k,j}^{(i)} \in \mathbb{R}^{b_k}, \quad \text{for } j = 1, \ldots, q_k, \text{ and } i = 1, \ldots, r,$$

we apply the reduction $S_{j_1}^T$ on both sides of (13), to obtain

$$\begin{aligned}
\mathbf{v}_{1,j_1}^{(i)} &\overset{1}{=} \sum_{j_2=1}^{q_2} \cdots \sum_{j_d=1}^{q_d} (I, \mathbf{v}_{2,j_2}^{(i)}, \ldots, \mathbf{v}_{d,j_d}^{(i)})^T \cdot (S_{j_1}, S_{j_2}, \ldots, S_{j_d})^T \cdot \mathcal{A} \\
&\overset{1}{=} \sum_{j_2=1}^{q_2} \cdots \sum_{j_d=1}^{q_d} (I, \mathbf{v}_{2,j_2}^{(i)}, \ldots, \mathbf{v}_{d,j_d}^{(i)})^T \cdot \mathcal{A}_{j_1,j_2,\ldots,j_d}. \qquad \text{(Ref. M3B)}
\end{aligned}$$

The above equation immediately entails a memory-efficient procedure for computing a k-MTVP: one needs to compute the k-MTVPs with each of the $\prod_{j=1}^d q_j$ subtensors $\mathcal{A}_{j_1,\ldots,j_d}$ and their corresponding subvectors $\{\mathbf{v}_{j,i_j}^{(i)}\}_{j \neq k}$, and add these contributions to the resulting vector $\mathbf{v}_{k,i_k}^{(i)}$. This idea is visualized in Figure 1 For computing the k-MTVPs with the subtensors efficiently, both Ref. M1 and Ref. M3 qualify. As default implementation, we suggest the latter because of the considerations about the memory consumption at the end of section 3.2 and the observation that these fringe cases may arise frequently with blocking, namely whenever $n_i = b_i q_i + 1$. We refer to this implementation with blocking as Ref. M3B.

Explicit subtensor storage. Blocking was proposed chiefly to combat the exceeding memory requirements of both Ref. M1 and Ref. M3. While the proposed method is clearly effective in this respect, care must be taken when implementing it in order to attain high throughput. The key problem is that the elements of the subtensor do not appear in consecutive memory, so that the successive contractions optimization does not apply: we must (temporarily) store the subtensors according to the canonical vectorization if Ref. M3 is to be applied to every subtensor. In the case of matrices, the idea of packing submatrices into consecutive memory fitting in the highest

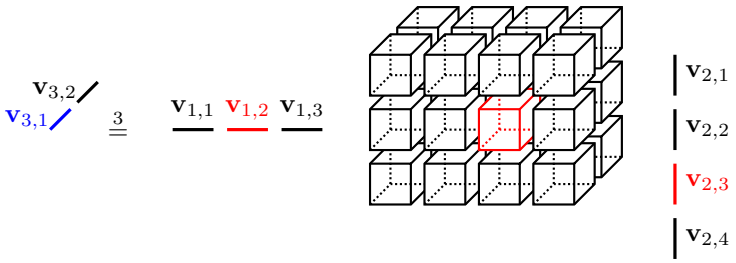

FIGURE 1. A schematic representation of $\mathbf{v}_3 = (\mathbf{v}_1, \mathbf{v}_2, I)^T \cdot \mathcal{A}$. Indicated in color, is the partial contribution $(\mathbf{v}_{1,2}, \mathbf{v}_{2,3}, I)^T \cdot \mathcal{A}_{2,3,1}$ which should be added to $\mathbf{v}_{3,1}$.

level of cache memory was demonstrated to be very effective on modern computer systems for matrix multiplication by Goto and van de Geijn (2008) and one may expect that this technique would be effective for higher-order tensors as well. In our implementation, however, we elected to explicitly store the subtensors in consecutive memory; that is, the elements of a subtensor all appear successively in memory, followed by all elements of the next subtensor, and so on. The elements of the subtensor are linearized according to the canonical vectorization. We will refer to this scheme as *subtensor storage*. It fits within a general tendency for more algorithmic abstraction in recent research into high-performance matrix algorithms without sacrificing performance, see, e.g., Gustavson (1997); Gunnels et al. (2001); Quintana-Ortí et al. (2009, 2012). We recall that Schatz et al. (2013) also proposed a variant of subtensor storage for symmetric tensors in their quest for efficient algorithms computing a symmetric multilinear multiplication. Our motivation for choosing explicit subtensor storage is twofold. First, in the setting where the k-MTVP is repeatedly computed it is beneficial to convert the tensor to subtensor storage prior to these operations to improve the memory access pattern to elements of the tensor $\mathcal{A}$ and reduce the number of index calculations. Our preliminary experiments confirmed that this technique improves the throughput with respect to the alternative of packing the data into the cache memory on demand. Nevertheless, the performance gain was usually less than 15%, and the advantage wanes as the number of vectors r, and thus computational intensity, is increased. Our second motivation is related to extensions of the k-MTVP to other important settings, namely, parallel and out-of-core algorithms, wherein we believe subtensor storage is unavoidable to attain good performance. We are currently investigating a parallel implementation that exploits subtensor storage, but this falls outside of the scope of the current paper.

With subtensor storage, the order in which the subtensors are linearized influences the access pattern to the matrices V_j, $j \neq k$. Linearizing the subtensors in a random order may cause poor cache reuse of aforementioned matrices and, hence, lead to worse throughput. It is therefore advisable to process the subtensors in some predetermined order, so as to improve the odds that the hardware prefetching mechanisms in modern processing units perform well. Considering the subtensors as (formal) elements of a $q_1 \times \cdots \times q_d$ array $\mathcal{D}$, we propose to order the subtensors according to the order specified by the canonical vectorization of $\mathcal{D}$. This linearization is surely not optimal with respect to the number of cache misses sustained when accessing parts of V_j and more complicated schemes such as the Morton, or Z-curve, ordering (Morton, 1966) can be pursued. We are convinced that in typical scenarios, however, the matrices $\{V_j\}$ are sufficiently small so as not to materially increase the number of cache misses. Consider, for instance, the storage requirements of the 500×450 matrices arising in a k-MTVP with a $500 \times 500 \times 500$ tensor, which requires 1GB of memory, and $r = 450$ vectors. The computer system used in our experiments would be able to hold all three matrices concurrently in its cache. Because we

do not expect a significant benefit from employing more sophisticated linearization schemes, we leave the issue as an open question.

In all subsequent discussions, *it is assumed that Ref. M3B is implemented for tensors that are stored by subtensors.*

Time and space complexity. By definition, it follows that the time complexity is $\prod_{j=1}^{d}(n_j/b_j)$ times $\mathrm{ops}_{\mathrm{M3}}(k, r, b_1, \ldots, b_d)$, which after reorganizing terms may be verified to equal

$$\begin{aligned}&\mathrm{ops}_{\mathrm{M3B}}(k, r, n_1, \ldots, n_d, b_1, \ldots, b_d)\\ &\quad= \mathcal{O}\Bigg(2r\prod_{j=1}^{d} n_j\Bigg(\sum_{i=1}^{\imath}\prod_{j=1}^{i-1} b_{s_j}^{-1} + b_k\prod_{j=1}^{\jmath-1} b_{s_j}^{-1}\sum_{i=\jmath}^{\ell}\prod_{j=i+1}^{\ell} b_{s_j}^{-1}\Bigg) + r(d-\ell+b_k)\prod_{i=1}^{d}(n_i/b_i)\Bigg) \text{ flops.}\end{aligned}$$

Again one notes that the dominant term is equal to $2r\prod_{j=1}^{d} n_j$, which is the same as in Ref. M1 and Ref. M3. Nevertheless, the lower-order terms are larger in the case of Ref. M3B and depend on the block sizes. Consider, for clarity, the d-MTVP with Ref. M3, respectively Ref. M3B, for an $n_1 \times \cdots \times n_d$ tensor wherein $n_i > 1$ for all i; then, the number of operations are, respectively,

$$\mathrm{ops}_{\mathrm{M3}} = 2r\prod_{j=1}^{d} n_j\big(1 + n_1^{-1} + n_1^{-1}n_2^{-1} + \cdots + n_1^{-1}\cdots n_{d-1}^{-1}\big), \text{ and}$$

$$\mathrm{ops}_{\mathrm{M3B}} = 2r\prod_{j=1}^{d} n_j\big(1 + b_1^{-1} + b_1^{-1}b_2^{-1} + \cdots + b_1^{-1}\cdots b_{d-1}^{-1}\big).$$

From these expressions it is clear that it is advantageous to maximize $b_1 \leq n_1$. In general the optimal order of the factors of the tensor is such that $b_{s_1} \geq \cdots \geq b_\imath \geq b_\ell \geq \cdots \geq b_\jmath$, and the block size that minimizes the number of operations is $b_i = n_i$ for all i. Of course, the latter suggestion would negate the advantage of the reduced memory consumption; some compromise should be sought. This issue is investigated more thoroughly in section 4.2.1.

The memory-efficiency of Ref. M3B for computing a k-MTVP is clear from the space complexity of Ref. M3: the number of values stored is $\mathcal{O}((1 + \frac{r}{b_{s_1}})\prod_{i=1}^{d} b_i)$ for a k-MTVP with $k \neq 1$, and $\mathcal{O}((1 + \frac{r}{b_{s_\ell}})\prod_{i=1}^{d} b_i)$ values otherwise. By appropriately choosing the block size, this algorithm can operate within any memory restrictions imposed by the user. This constitutes the major advantage of Ref. M3B over the other approaches.

4. Numerical experiments

In this section, experiments will be performed with two experimental setups. We developed both a Matlab implementation using the TensorToolbox v2.6 (Bader and Kolda, 2012), and a C++ implementation founded on the matrix library Eigen3 (Guennebaud et al., 2010). We consider the C++ implementation to be reasonably optimized.[9]

The Matlab code will illustrate the simplicity of some of the proposed optimizations along with the performance gains that may be expected. In general, the codes we developed rely extensively on built-in matrix routines that are implemented through optimized and compiled libraries. However, as our experiments also demonstrate, some of the Matlab built-in routines, e.g., the `permute` function manipulating the ordering of elements in multidimensional arrays, do not always guarantee maximal performance. In algorithms relying on such operations, the relative performance of the considered optimizations may substantially differ in the C++ and

[9]Our choice of Eigen3 was based entirely on the familiarity of the authors with this package. We believe that results very similar to those presented would be obtained with other optimized libraries such as Blaze or MKL. A detailed comparison of these libraries is, however, out of the scope of this work.

Matlab implementations. We regard the C++ implementation as the most reliable indicator of true performance in these cases.

The experiments with the Matlab code were performed in Matlab 7.9.0 on a computer system consisting of an Intel Core i5 M560 processing unit clocked at 2.67GHz and 4GB of main memory. As we focus on sequential performance in this paper, Matlab was instructed to use only one computational thread by starting it with `-singleCompThread`. Experiments with the C++ code were conducted on a computer system consisting of one Intel Core2Duo E8500 dual-core processor clocked at 3.16GHz (with dynamic CPU frequency scaling disabled), 6MB L2 cache memory, and 3.8GB of main memory. To ensure optimal testing conditions, the executable is called with `numactl --physcpubind=+0` to pin its execution on the first physical processing core; additionally, after allocating all memory our program requires, an `mlock` system call is made from the C++ code, requesting that the current memory pages used by the executable are retained in the system's main memory during its execution. The C++ code was compiled with the flags `-O3`, `-std=c++0x`, `-msse4.1`, `-fwhole-program`, `-march=native`, `-funroll-loops`, `-malign-double` and `-fipa-pta` using the GCC v4.7.1.

In some of our experiments we will summarize the performance of k-(M)TVPs for all $k = 1, \ldots, d$ by aggregating the results of the individual operations; we denote this aggregated operation by $*$-(M)TVP. This operation occurs frequently in practice in computing CP decompositions by gradient-based optimization algorithms, where every k-MTVP yields a different component of the gradient, all of which are simultaneously needed in one step of the optimization procedure.

4.1. **The single vector tensor-vector product.**

4.1.1. *A comparison of the reference implementations.* As the k-TVP is a memory-bound operation, we expect that eliminating the explicit calculation of unfoldings by Ref. S3 is effective for reducing the execution time. This hypothesis is investigated here first for the Matlab implementation because we may expect the gain to be most significant here. For the sake of reproducibility and as an illustration of what is currently prevalent, we selected the `mttkp` routine in the Tensor Toolbox as implementation of Ref. S1, `ttv` as implementation of Ref. S2, and the Matlab code in section 2.3 as implementation of Ref. S3.[10] The total execution time of 5000 k-TVPs with one random $n \times n \times n$ tensor and one random set of vectors, where $n = 25, 50, 75, \ldots, 250$ and $k = 1, 2, 3$ was measured. The execution times were then normalized, for every n, with respect to the execution time of a 1-TVP computed with Ref. S1. The results are shown in Table 1.

Considering first Ref. S1 and Ref. S2, it can be seen in Table 1 that the execution times of these methods are roughly in line with the theoretical prediction for large n; both perform equally well. However, for n up to 125, the performance of Ref. S2 is worse than the theory would suggest. Therefore, we repeated these experiments with the C++ implementation where the normalized execution times of Ref. S2 for a 1-TVP, for example, were respectively 0.82, 0.77, 1.28, 0.94, and 1.03, which is in closer correspondence with the theory. An item that stands out in the table is the considerable performance difference between the 1-TVP and a k-TVP in the other modes; the execution time of the latter can be up to one order of magnitude higher. This discrepancy stems from the Matlab built-in `permute` function, which is used by both `mttkrp` and `ttv` to rearrange the factors of the tensor such that they appear in the correct order for constructing the unfolding, respectively, for performing a RTL contraction. A 1-TVP incurs no cost since the factors are already in the correct order.

[10] The routine `ttv` first reorders the factors of the tensor using the Matlab built-in `permute` function so that the kth factor appears first in the reordered tensor, and then performs the RTL contraction detailed in section 2.3. Consequently, the number of memory operations is comparable to that of Ref. S1. The main observation will nevertheless be clear: avoiding data reorderings is far more efficient.

k	Method	n									
		25	50	75	100	125	150	175	200	225	250
1	Ref. S1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	Ref. S2	3.70	3.44	2.04	1.48	1.24	1.16	1.02	1.08	1.03	1.09
	Ref. S3	3.12	2.98	1.87	1.39	1.19	1.14	1.01	1.07	1.03	1.08
2	Ref. S1	**1.16**	**2.37**	4.33	5.97	7.31	9.05	8.84	6.33	9.84	8.57
	Ref. S2	3.84	5.68	5.36	6.54	7.57	9.26	8.85	6.41	9.86	8.66
	Ref. S3	3.14	3.08	**2.00**	**1.49**	**1.29**	**1.24**	**1.06**	**1.12**	**1.00**	**1.05**
3	Ref. S1	**1.17**	**2.72**	6.16	5.98	7.61	8.11	9.22	7.52	10.11	7.44
	Ref. S2	3.85	6.07	7.19	6.54	7.88	8.33	9.26	7.60	10.15	7.59
	Ref. S3	3.15	3.07	**2.00**	**1.49**	**1.29**	**1.24**	**1.06**	**1.12**	**1.00**	**1.05**
*	Ref. S1	**3.33**	**6.09**	11.49	12.95	15.92	18.16	19.06	14.85	20.95	17.01
	Ref. S2	11.39	15.19	14.59	14.56	16.69	18.75	19.13	15.09	21.04	17.34
	Ref. S3	9.41	9.13	**5.87**	**4.37**	**3.77**	**3.62**	**3.13**	**3.31**	**3.03**	**3.18**

TABLE 1. Normalized execution times for 5000 k-TVPs with a random tensor of dimension $n \times n \times n$ performed with the reference implementations in Matlab.

One observes that the execution time of Ref. S3 is largely independent of the mode in which the vector is generated and, crucially, it outperforms the other implementations for $k \neq 1$ by a large margin: it can be up to 6 times faster for large tensors. In the common case where a $*$-TVP is repeatedly computed, the performance of Ref. S3 is superior over the other implementations, and we are convinced that it is the most suitable implementation for Matlab. One may expect that Ref. S3 becomes progressively more advantageous over the other methods if the order of the tensor is increased.

In Table 2, the performance advantage of Ref. S3 over Ref. S1 is reaffirmed with the C++ implementation. Herein, Ref. S1 is competitive for both 1-TVPs and 3-TVPs because no unfoldings are explicitly computed. In the Matlab experiments, the performance of the 3-TVP was also hampered, because the `permute` function will explicitly construct and allocate new memory for $\mathcal{B}_{(3)} = \mathcal{B}^T_{(1,2;3)}$. The data highlight the harmful effect of computing unfoldings on the performance of Ref. S1: with the exception of the first set of experiments, computing a 2-TVP is at least 3 times as expensive as a TVP in the first or last mode.

4.1.2. *Computational intensity.* The first matrix-vector multiplication in Ref. S3 usually strongly dominates the cost of the other operations. Indeed, these lower-order terms add at most a fraction of $\frac{d-1}{n_1}$, if $k \neq 1$, and $\frac{d-1}{n_d}$, otherwise, to the total number of operations relative to the cost of the matrix-vector product. This would suggest to maximize n_1 for limiting the cost of a k-TVP, if $k \neq 1$, and similarly for n_d and a 1-TVP. From the Ω columns in Table 2, it can be verified that the number of operations performed is positively correlated with the total execution time in every set of experiments. This does not fully explain the difference in execution times, however. For instance, in the set of experiments with the $5000 \times 75 \times 25$ tensor the worst ordering of the factors has (only) 2.65% more operations than the best ordering, yet the former is 52.5% slower than the latter.

4.1.3. *Order of modes.* The execution time of Ref. S3 depends on the order of the factors of the tensor for two important reasons: the number of operations can be different, and the throughput of the matrix-vector product depends on the shape of the matrix. In the previous subsection, we already remarked that the former only accounts for a fraction of the observed differences in Table 2.

	n_1	n_2	n_3	T_1	τ_1	Ω_1	T_2	τ_2	Ω_2	T_3	τ_3	Ω_3	T_{tot}
Ref. S1	5	5000	400	28.4	845	2.40	51.2	390	2.00	13.3	1501	2.01	92.9
Ref. S3	5000	5	400	13.3	1502	2.01	13.4	1488	2.00	13.4	1492	2.00	40.2
	400	5	5000	13.5	1480	2.00	13.6	1474	2.01	13.6	1469	2.01	40.7
	400	5000	5	31.8	753	2.40	13.6	1474	2.01	13.5	1485	2.01	58.9
	5000	400	5	32.2	744	2.40	13.4	1488	2.00	13.4	1494	2.00	59.1
	5	400	5000	13.5	1479	2.00	44.1	544	2.40	43.9	546	2.40	101.5
	5	5000	400	13.3	1504	2.01	45.4	528	2.40	45.5	527	2.40	104.2
Ref. S1	5000	75	25	12.4	1510	2.00	56.9	334	2.03	17.8	1096	2.08	87.1
Ref. S3	5000	25	75	13.7	1385	2.03	12.4	1512	2.00	12.4	1515	2.00	38.5
	5000	75	25	19.0	1025	2.08	12.6	1485	2.00	12.6	1489	2.00	44.2
	75	25	5000	12.5	1495	2.00	15.9	1191	2.03	16.0	1187	2.03	44.5
	75	5000	25	18.9	1028	2.08	16.1	1181	2.03	16.0	1189	2.03	51.0
	25	75	5000	12.6	1486	2.00	22.7	859	2.08	22.9	853	2.08	58.1
	25	5000	75	13.8	1379	2.03	22.6	864	2.08	22.4	869	2.08	58.7
Ref. S1	1000	200	50	14.1	1420	2.00	57.8	348	2.01	17.1	1194	2.04	88.9
Ref. S3	1000	50	200	13.4	1494	2.01	13.9	1439	2.00	13.9	1437	2.00	41.3
	200	50	1000	13.6	1472	2.00	14.5	1388	2.01	14.5	1387	2.01	42.5
	1000	200	50	18.1	1126	2.04	13.9	1439	2.00	13.9	1440	2.00	45.9
	200	1000	50	18.2	1120	2.04	14.2	1414	2.01	14.2	1414	2.01	46.6
	50	1000	200	13.4	1495	2.01	18.0	1134	2.04	18.1	1124	2.04	49.5
	50	200	1000	13.5	1476	2.00	18.2	1119	2.04	18.3	1115	2.04	50.0
Ref. S1	500	250	125	21.4	1465	2.00	91.1	344	2.01	22.4	1408	2.02	134.8
Ref. S3	500	125	250	21.3	1471	2.01	20.7	1512	2.00	20.7	1512	2.00	62.7
	500	250	125	22.9	1377	2.02	20.8	1501	2.00	20.8	1505	2.00	64.5
	250	125	500	22.1	1416	2.00	21.8	1435	2.01	21.9	1434	2.01	65.8
	250	500	125	22.9	1375	2.02	21.8	1435	2.01	21.8	1435	2.01	66.6
	125	250	500	22.1	1418	2.00	24.7	1275	2.02	24.5	1283	2.02	71.3
	125	500	250	21.3	1471	2.01	25.4	1238	2.02	24.5	1282	2.02	71.3
Ref. S1	200	400	300	34.0	1418	2.01	129.2	372	2.01	31.6	1522	2.01	194.8
Ref. S3	400	200	300	31.5	1525	2.01	31.3	1536	2.01	31.5	1526	2.01	94.4
	300	200	400	30.8	1559	2.01	31.7	1515	2.01	31.8	1512	2.01	94.4
	200	300	400	30.7	1564	2.01	33.1	1454	2.01	33.2	1451	2.01	97.1
	200	400	300	31.0	1554	2.01	33.2	1452	2.01	33.1	1455	2.01	97.3
	300	400	200	34.1	1414	2.01	31.7	1517	2.01	31.7	1516	2.01	97.5
	400	300	200	34.8	1384	2.01	31.4	1530	2.01	31.4	1533	2.01	97.6
Ref. S1	300	300	300	36.3	1489	2.01	150.6	359	2.01	35.4	1528	2.01	222.3
Ref. S3	300	300	300	35.0	1547	2.01	35.8	1511	2.01	35.9	1507	2.01	106.7

TABLE 2. The execution time in seconds (T_k), throughput in Mflop/s (τ_k), computational intensity (Ω_k), and the total execution time $T_{\text{tot}} = T_1 + T_2 + T_3$ of 1000 k-TVPs for random $n_1 \times n_2 \times n_3$ tensors. These results are displayed for Ref. S3 as well as for the order of the modes that yielded the least total execution time for Ref. S1. Ω_k is defined as the average number of operations performed per element in the tensor. These results were obtained with the C++ implementation.

The prime observation concerns the order of the factors that minimizes the execution time for Ref. S3: from the data in Table 2 one learns that for every shape tested, the order that minimizes the execution time was $n_1 \geq n_3 \geq n_2$. We attribute this to two (empirical) observations. First, note that if $k = 2, 3$, we start with an LTR contraction, and otherwise with an RTL contraction. The matrix-vector product in the first step of either an LTR or RTL contraction contributes the

	n_1	n_2	n_3	n_4	T_1	τ_1	T_2	τ_2	T_3	τ_3	T_4	τ_4	T_{tot}
Ref. S1	5	500	10	100	5.7	1.05	13.1	0.38	15.5	0.35	3.3	1.51	37.7
Ref. S3	500	10	5	100	3.3	1.52	3.6	1.40	3.6	1.39	3.6	1.39	14.1
Ref. S1	200	75	50	25	26.9	1.40	110.5	0.34	115.7	0.33	34.9	1.12	287.9
Ref. S3	200	50	25	75	35.9	1.06	26.2	1.44	26.1	1.44	26.2	1.44	114.4
Ref. S1	12	50	100	25	2.8	1.17	8.6	0.36	8.8	0.34	2.0	1.57	22.2
Ref. S3	100	25	12	50	2.0	1.56	2.3	1.29	2.4	1.29	2.3	1.29	9.0
Ref. S1	40	50	60	30	6.7	1.11	21.1	0.35	22.5	0.33	5.2	1.44	55.4
Ref. S3	60	40	30	50	4.9	1.50	6.0	1.21	6.0	1.21	6.0	1.21	23.0
Ref. S1	60	60	60	60	21.1	1.25	76.4	0.35	80.6	327	22.0	1197	200.0
Ref. S3	60	60	60	60	24.2	1.09	21.8	1.21	21.8	1207	21.8	1207	89.7

TABLE 3. The execution time in seconds (T_k), throughput in Gflop/s (τ_k), and the total execution time $T_{\text{tot}} = T_1 + T_2 + T_3 + T_4$ of 1000 k-TVPs for random $n_1 \times n_2 \times n_3 \times n_4$ tensors. These results are displayed for the order satisfying $n_1 \geq n_4 \geq n_2 \geq n_3$ for Ref. S3 and for the order of the modes that yielded the least total execution time for Ref. S1. These results were obtained with the C++ implementation.

dominant term in the time complexity analysis of Ref. S3. As a consequence, its throughput will roughly determine the throughput of the entire k-TVP. We observed empirically that the throughput of Eigen's implementation of the matrix-vector product is higher whenever the shape of the matrix is more square and if none of the sizes is very small, say less than 200. Consider, for instance, the 3-TVPs of all possible orderings of the modes of an $5000 \times 75 \times 25$ tensor, displayed in the second set of experiments in Table 2: the throughput of the matrix-vector product with a 5000×1875 matrix and vector of length 5000 is approximately 1.5Gflop/s, as seen in the first two rows of Ref. S3; of the product with a 75×125000 matrix and compatible vector it is approximately 1.188Gflop/s, as seen in the next two rows; and of the product with a 25×375000 matrix it is only 0.861Gflop/s, as displayed in the final two rows. Consequently, for an LTR contraction, it is beneficial that n_1 is the largest mode, and, conversely, n_d should be the largest mode for attaining optimal performance with an RTL contraction. Second, by the previous observation and the fact that our implementation starts with an LTR contraction whenever $k \neq 1$, it follows that the total execution time will be less if $n_1 \geq n_d$ rather than the other way around.

Based on the above considerations, one arrives at the conclusion that it is beneficial to rearrange the factors in such a way that $n_1 \geq n_d \geq n_2 \geq \ldots \geq n_{d-1}$, provided that the cost of this rearrangement may be amortized over the execution of multiple TVPs. Looking at the total execution times in Table 2, it can be seen that rearrangement is most beneficial if there exist large differences in the size of the modes and some of them are very short, as in the first three sets of experiments.

As a final illustration of the advantage of Ref. S3 over Ref. S1 we consider some experiments for fourth-order tensors in Table 3. We display only the order yielding the least execution time for Ref. S1, and the order corresponding to the aforementioned heuristic for Ref. S3. The heuristic performs well, and also corresponded to the best ordering in all of these examples. Note again the negative impact of constructing unfoldings on the performance of Ref. S1: only k-TVPs with $k = 1, d$ are competitive with Ref. S3; for the other modes, Ref. S1's execution time is at least thrice Ref. S3's. The speedup of Ref. S3 over Ref. S1 when performing a k-TVP for every k is at least two in every case.

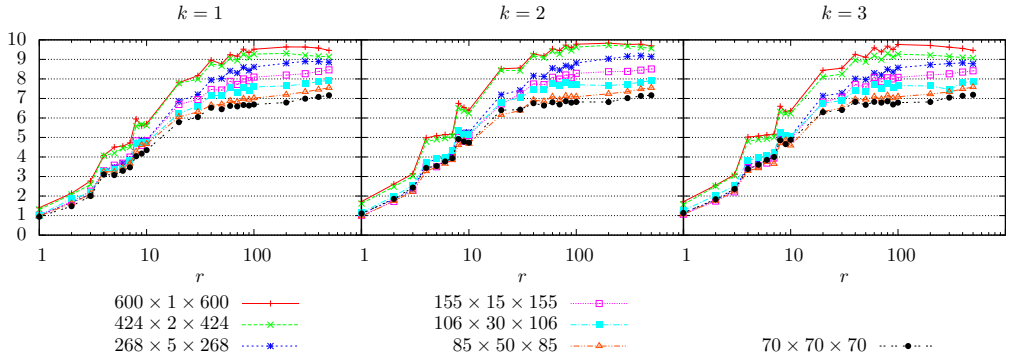

FIGURE 2. Effective throughput in Gflop/s of the k-MTVP implemented by Ref. M3 in function of the number of vectors r.

4.2. **The multiple vector tensor vector product.**

4.2.1. *Subtensor sizes.* Following Goto and van de Geijn (2008), it seems logical to pursue a division of the tensor into subtensors for Ref. M3B so that the individual subtensors fit into the highest level of the cache memory hierarchy.[11] On the other hand, in the interest of software reuse it is recommended to employ existing thoroughly optimized matrix multiplication routines—such as, e.g., ATLAS, GotoBLAS, Intel MKL, Eigen and Blaze—for performing the first step of a k-MTVP with one of the subtensors. As the performance of these routines should not degrade significantly if the dimensions of the matrix exceed the size of the cache memory, one may already expect that performance ought not be compromised in this case because it would be handled by the matrix library. Our preliminary numerical experiments indeed confirmed that if the subtensor does not fit entirely in the cache memory then this did not reduce performance significantly. In one set of experiments we tricked the matrix library into believing that the level 2 cache was infinitely large[12] effectively disabling the mechanism that packs parts of the matrix into consecutive positions in the cache memory; in this case, it was observed that the throughput did degrade significantly whenever the matrix could not be fitted into the cache memory. These experiments support our conclusion that whenever one uses an optimized matrix library, *it is from the viewpoint of expeditious execution unnecessary to restrict oneself to subtensor sizes fitting entirely in the cache memory.*

In contrast to the previous remark, *the shape* of the subtensors with which one multiplies in Ref. M3B was determined to impact performance strongly. For investigating this, we performed 100 k-MTVPs with r vectors using Ref. M3 for several tensors. The shapes of these tensors were chosen so that every tensor contains approximately 350000 nonzero elements, hence requiring about 2.9MB of memory, which fits entirely in the cache memory.[13] We have chosen to limit ourselves to such tensors primarily to exclude the effect of the packing mechanism employed by the matrix library. In all of the remaining experiments, the shape of the subtensor division for

[11] Beware that these recommendations apply only to the case of a single processing unit, where we may assume exclusive access to the entire cache hierarchy.

[12] The Eigen (Guennebaud et al., 2010) library that we used is based on the principles in (Goto and van de Geijn, 2008) and allows the user to specify the cache sizes by calling `setCpuCacheSizes`.

[13] We may limit ourselves to assessing the performance of Ref. M3 for a given shape to investigate the performance of Ref. M3B with subtensors of that shape because the performance of the latter is determined nearly completely by the former's performance on tensors of that shape.

Ref. M3B is chosen so that the subtensor fits entirely in the highest level of the cache memory hierarchy, which in our experimental setup was 6MB large. The results of the experiments are summarized in Figure 2, wherein we plot for every shape the *effective throughput*—which we define as the approximate number of operations $2r\prod_{i=1}^{d} n_i$ divided by the total execution time—in function of the number of vectors r. Notice that the number of flops of Ref. M3 is r times that of Ref. S3 so that the observations in section 4.1.3 are still valid, and, in particular, that the suggested heuristic for ordering the factors may still be expected to perform well. For this reason, we did not include the results of the other permutations of the factors in Figure 2 as they were in line with our predictions.[14] The effective throughput was preferred over the throughput, because the actual number of operations performed can be a gross overestimation of the number of *useful* operations—for instance, as in the fringe case discussed in section 3.2. The number of useful operations is $2r\prod_{i=1}^{d} n_i$, because for each of the r k-TVPs all the elements of the tensor will be involved in at least one multiplication and one addition in each of the reference implementations considered. We do not know whether it is theoretically possible to improve upon this cost for general tensors. Nonetheless, for each of the considered implementations this operation count is a lower bound, so that the effective throughput always underestimates the actual throughput.

As a first observation concerning Figure 2, we note that the effective throughput is close to the peak performance of the machine. Using SSE4.1 extensions, the computer system can complete up to 4 double-precision floating-point operations per clock cycle (two packed additions and two packed multiplications), so that the theoretical peak performance is 12.64Gflop/s. The k-MTVP implemented with Ref. M3 with $600 \times 1 \times 600$ tensors attains up to 78% of this theoretical peak performance for every k. The same observation holds for the shapes $1\times600\times600$ and $600\times600\times1$ as well, by virtue of the way we handle the fringe case discussed in section 3.2. We omitted their results in Figure 2 because their graphs virtually coincided with that of $600 \times 1 \times 600$.

The matrix products in the first step of the k-MTVP that resulted in the best performance, regardless of the value of r and k, were in order of decreasing performance 600×600, 424×848, 268×1340, 155×2325, 106×3180, 85×4250, and 70×4900, as we can deduce from Figure 2. Continuing the sequence with data obtained from different orderings of the factors of the shapes tested in Figure 2, we found 50×7225, 30×11236, 15×24025, 5×71824, and 2×179776, again in order of decreasing performance. We thus observe that matrix products with more square shapes typically result in better throughput with the Eigen matrix library *in the considered setting where the matrices appearing in the product all fit in the cache memory.* This behavior is in line with the theoretical prediction that the number of operations per data transfer is higher for square shapes fitting in the cache memory, which, according to (Goto and van de Geijn, 2008, Section 4.1), applies under some assumptions for certain simplified computer architectures and provided that the matrix product is implemented using packing.

Good performance in absolute terms can be obtained for a k-MTVP provided that the shape of the subtensors is chosen so that the required matrix product in the first step involves an approximately square matrix that is as large as possible while still fitting entirely in about half of the cache memory. Note that some room should be left in the cache memory for the matrix with which one multiplies, hence the suggestion to use only half of the available space. We remark that it is possible to subdivide the tensor into subtensors of size $b_1 \times \cdots \times b_d$ so that for a k-MTVP a multiplication with an approximately square matrix arises for every k. With the usual definitions from section 3.2, and considering that the matrix is of the shape $b_{s_1} \times \prod_{j=s_1+1}^{d} b_j$ if $k \neq 1$ and $b_{s_\ell} \times \prod_{j=1}^{s_\ell-1} b_j$ otherwise, there is but one possibility to obtain (approximately) square

[14]For the shapes tested, if the first factor has the shortest length, then the performance deteriorates for k-MTVPs with $k \neq 1$, and equally so for $k = 1$ if the last factor is the shortest.

matrices for *every* k-MTVP: choose $S = \langle s_1, s_2 \rangle$ with $b_{s_1} \approx b_{s_2}$ and all other $b_i = 1$; that is, *subdivide the tensor into matrix slices.*

Restricting oneself to slicing rather than blocking benefits the operation count: the time complexity estimate of Ref. M3B reduces to

$$2r(1+\zeta)\prod_{j=1}^{d} n_j + r(d-\ell+b_k)\prod_{j=1}^{d}\frac{n_j}{b_j} \quad \text{where } \zeta = \begin{cases} 0 & \text{if } k = s_1 \text{ or } k = s_2 \\ b_{s_1}^{-1} & \text{if } k > s_1 \\ b_{s_2}^{-1} & \text{if } k < s_1 \end{cases}. \tag{14}$$

If we may assume that b_{s_1} and b_{s_2} have been chosen maximally to fit within a given amount of memory and that $r(d-\ell+b_k)b_{s_1}^{-1}b_{s_2}^{-1} \ll 1$, i.e., the second term is dominated by the first, then ζ appears to be the optimal value for a given allowed memory consumption, i.e., for constant $\prod_{j=1}^{d} b_j$. Changing any of the $b_i = 1$ to a larger value must proportionally decrease the value of b_{s_1} or b_{s_2} or a combination thereof, which would only serve to increase the value of ζ for at least one k-MTVP while the second term remains unaffected. Note that by the same reasoning $b_{s_1} \neq 1$ with all other $b_i = 1$ may seem even more attractive. However, the matrix product would be replaced by a matrix-vector product that will attain lower throughput. Additionally, the second term in the operation count will then contribute at least $r\prod_{j=1}^{d} n_j$ operations if $k = s_1$, i.e., a relative increase of nearly 50%. This is not the case with slicing, where the second term will only add $rb_{s_2}^{-1}\prod_{j=1}^{d} n_j$ operations if $k = s_1$, respectively $rb_{s_1}^{-1}\prod_{j=1}^{d} n_j$ if $k = s_2$.

Slicing with $n_1 \times n_2 \times 1 \times \cdots \times 1$ subtensors may be particularly interesting in combination with the canonical vectorization of the tensor. With this subdivision it is namely unnecessary to store the tensor with subtensor storage while still employing high-performance matrix routines. That is because all required elements of the subtensor reside in consecutive memory positions, so that the successive contractions optimization applies. In addition, note that $n_1 \times n_2 \times 1 \times \cdots \times 1$ subtensors would also minimize the number of operations in (14) provided that n_1 and n_2 are the largest two sizes. More generally, blocking with $b_1 \times b_2 \times 1 \times \cdots \times 1$ with $b_1 \leq n_1$ and $b_2 \leq n_2$ may also be expected to attain good performance without an explicit block storage order because the required subtensors correspond to submatrices of the $n_1 \times n_2 \times 1 \times \cdots \times 1$ subtensor division. As the latter appears consecutively in memory, it implies that the matrix multiplication appearing in the first step of Ref. M3 for a $b_1 \times b_2 \times 1 \times \cdots \times 1$ subtensor can be performed with a good implementation of the BLAS without allocating new memory, simply by providing the stride length n_1 to the appropriate routine.

Notwithstanding the benefits, slicing has its limitations: it requires at least two "large" modes. That is, a matrix product can attain near optimal performance only if *both* of the dimensions of the matrix are sufficiently large, as we can also deduce from Figure 2. If this requirement is met, we would propose the following heuristic: assuming the factors of the tensor have been reordered such that $n_1 \geq n_2 \geq \cdots \geq n_d$, choose $b_1 = n_1$ and $b_2 = n_2$ and all other $b_i = 1$ provided that an efficient matrix library is employed for computing the matrix product. Alternatively, one can perform the cache blocking explicitly by choosing $b_1 \approx b_2$ such that this matrix is as large as feasible while fitting in half of the cache memory; with our test setup 500×500 matrices would satisfy this requirement.

If only one of the dimensions is large but the other is small, slicing may be outperformed by other blocking strategies. In this case, assuming that the first factor is the largest, one can still obtain $d-1$ matrix products with near optimal performance and one less efficient product. This is due to our choice of starting with an LTR contraction when computing a k-MTVP with $k \neq 1$. In this case one should choose b_1 sufficiently large and $b_2 \cdots b_d \approx b_1$. We illustrate this suggestion in Figure 3, where we plotted the effective throughput for one $500 \times 40 \times 30 \times 20$ tensor, which was subdivided into subtensors of different shapes; the experiments were repeated

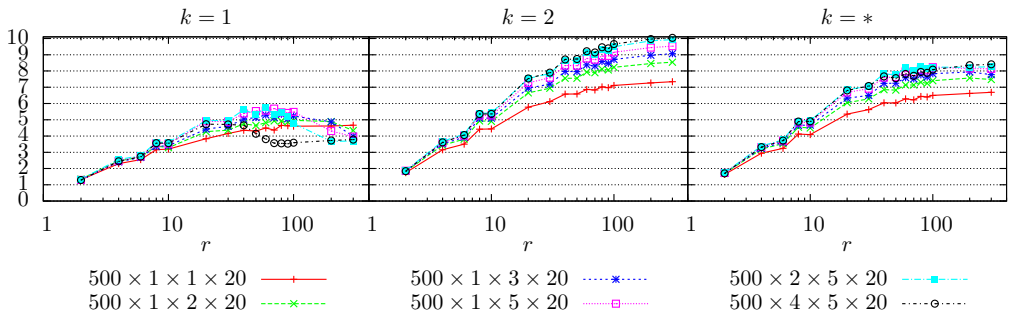

FIGURE 3. Effective throughput in Gflop/s of the k-MTVP for $k = 1, 2, *$ implemented by Ref. M3B for a $500 \times 40 \times 30 \times 20$ tensor for various shapes of the subtensor division.

100 times. In the figure, we only display the results for $k = 1, 2$ because the behavior of $k = 3, 4$ resembles closely the 2-MTVP. The performance of k-MTVPs with $k \neq 1$ can be observed to consistently increase by choosing subtensor sizes resulting in unfoldings with an increasing number of columns; we see that a 500×20 unfolding yields poor performance relative to a 500×200 or 500×400 unfolding. On the other hand, choosing larger subtensors penalizes the performance of the 1-MTVP with a large number of vectors as may be seen in the figure. This behavior occurs because the matrix product with the unfolded subtensor can increase the storage requirements of the resulting matrix to beyond the point where it can be contained in the cache memory, therefore inducing a performance hit. For instance, the $500 \times 4 \times 5 \times 20$ subtensor division, which requires 1.6MB of storage per subtensor, would yield a $500 \cdot 4 \cdot 5 \times r$ matrix after the first multiplication; hence, if $r > 75$ then this matrix no longer fits in the cache memory of the test machine. One observes in Figure 3 that performance indeed degrades for this subdivision from a peak at $r = 40$ (when about 82% of the cache is filled with the unfolded subtensor, the $500 \times r$ matrix to multiply with, and the resulting $10000 \times r$ matrix) to around 3.6Gflop/s from $r = 70$ onwards, at which point the resulting matrix cannot be contained in the cache. One may be surprised about the magnitude of the reduction of the throughput when the resulting matrix no longer fits in the cache memory; however, it is important to realize that further vector operations need to be performed with the aforementioned matrix. Consequently, it will have to be transferred back into the cache memory to perform only a very small number of operations. That is, the bandwidth from and to the memory will limit the attainable throughput for this last part of the computation of the MTVP with the subtensor. Regardless of the poor performance of the 1-MTVP, the $*$-MTVP will attain a decent throughput because a fraction of $\frac{d-1}{d}$ of the number of operations will be performed through the highly efficient k-MTVPs with $k \neq 1$. Based on the above observations, we suggest the following heuristic *if only one of the factors can be considered large*: choose $b_{s_1} \geq b_{s_\ell} \geq b_{s_2} \geq \cdots \geq b_{s_{\ell-1}}$ with b_1 sufficiently large and $b_2 \cdots b_\ell \approx b_1$. If no optimized matrix library is employed, the choice should additionally allow all matrices to reside within the cache memory.

We do not handle the case where all factors of the tensor are very small in this study; we believe that more elaborate techniques are required in order to attain truly high performance.

4.2.2. *Comparison of the reference implementations.* For comparing the performance of Ref. M1, Ref. M3 and Ref. M3B, we considered k-MTVPs for every k with several shapes of the tensor and repeated this 100 times. We have mostly limited our investigation to tensor shapes that satisfy

the heuristic proposed in section 4.1.3, as we have already argued that such an ordering of the factors is most beneficial for Ref. M3; this does not put Ref. M1 at a disadvantage, because its performance is quite insensitive to the ordering of the factors especially for larger r where the relative cost of computing unfoldings declines. For Ref. M3B we simply consider a subdivision of the tensor into $b_1 \times b_2 \times 1 \times \cdots \times 1$ subtensors, i.e., we slice the tensor, with $b_i = \min\{n_i, 500\}$, $i = 1, 2$, so that every subtensor occupies at most 2MB of consecutive memory, which fits entirely in the 6MB cache memory.[15] In Figure 4, we display the total effective throughput of a $*$-MTVP for several tensor shapes and number of vectors, and Figure 5 shows the corresponding additional memory consumption, i.e., the memory requirements in addition to the cost of storing the input tensor.

It is immediately clear from Figure 5 that the additional memory requirements of Ref. M3B, which are essentially determined by the user, dwarf the requirements of both Ref. M1 and Ref. M3. For most of the considered fourth-order tensors the difference is *three orders of magnitude*. Key here is that this advantage does not impair the execution time, as we may understand from the graphs in Figure 4; for all shapes Ref. M3B outperforms Ref. M1 by a few percentage points, and the former is competitive with Ref. M3 for virtually every shape. In the few cases where Ref. M3 manages to materially outperform Ref. M3B, that is, $1000 \times 64 \times 500$ and $210 \times 30 \times 24 \times 210$, this performance difference could have been anticipated; indeed, this ordering of the factors does not correspond to the heuristic suggested in section 4.2.1 for slicing. Changing the order of the factors so that we have, respectively, a $1000 \times 500 \times 64$ and $210 \times 210 \times 30 \times 24$ shape, improves the effective throughput of Ref. M3B to the level of Ref. M3 (corresponding to its best ordering of the factors).

Note that the performance of Ref. M1 degrades significantly for most of the tested fourth-order tensors, as we see in Figure 4. In addition, we observe that this phenomenon also arises for Ref. M3 in some cases. The explanation is straightforward: considering Figure 5 one can confirm that the degradation of the performance coincides with the memory consumption surpassing approximately 1GB of main memory, at which point it was completely saturated. We should mention here that Figure 5 reports the "theoretical memory consumption," i.e., the minimum amount necessary to contain the required values. In practice it is, however, not always beneficial to restrict oneself to this absolute minimum, for it would require one, in our implementation, to write a matrix multiplication routine that computes $B \leftarrow AB$ in place, which we believe would be more difficult to implement efficiently. Therefore, our practical implementation does not compute this product in place but rather allocates twice the theoretical minimum amount of memory. The successive contraction steps are then implemented efficiently by using a regular matrix product that alternates between the two buffers for storing the result. This approach is more wasteful in terms of memory consumption but it was determined to be much more efficient than always allocating and deallocating the correct amount of memory.

To conclude, we should point out that Figure 4 masks the differences in the performance of the various k-MTVPs. From our discussion up to this point it should be clear that the performance of a k-MTVP implemented by Ref. M1 is generally[16] determined by the magnitude of n_k: if it is small the performance will be bad, otherwise it will be good. No amount of permuting the factors of the tensor will change this. For Ref. M3 and Ref. M3B the situation is generally determined completely by the size of n_1 if $k \neq 1$, and by n_d otherwise. If n_1 is sufficiently large, performance will be good for all $k = 2, \ldots, d$, regardless of the size of the lengths of the corresponding modes

[15] In some of the examples better performance could be obtained by choosing a more general subdivision or a more suitable order of the factors as was explained in section 4.2.1, but we have chosen to limit ourselves to only one promising type of subdivision to illustrate its good performance on a range of different tensor shapes.

[16] More precisely, it is determined by the minimum of n_k and $\prod_{j \neq k} n_j$, but in the remainder we will always assume that the latter product is always larger.

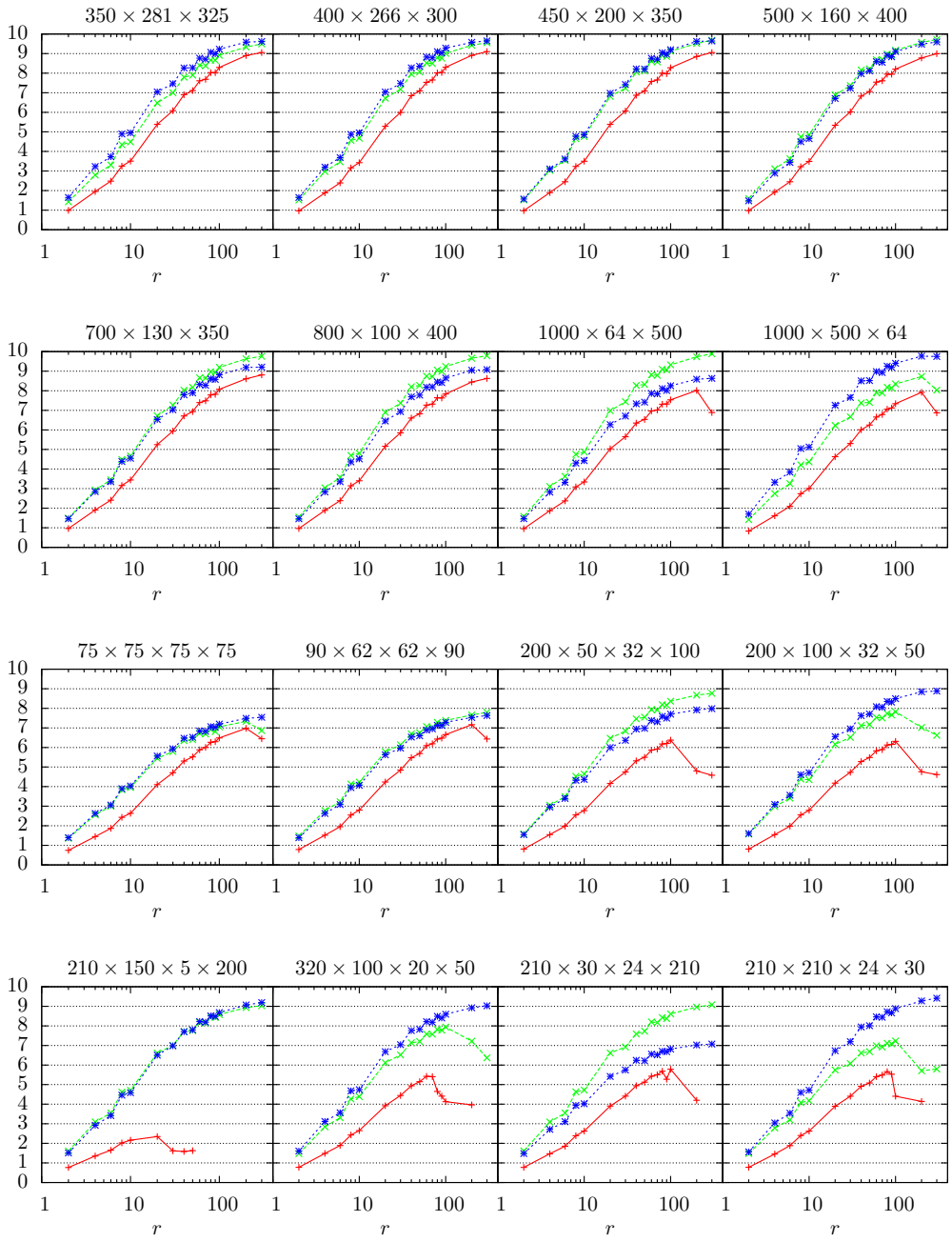

FIGURE 4. Effective throughput in Gflop/s of the $*$-MTVP with r vectors for Ref. M1 (—+—), Ref. M3 (----×----), and Ref. M3B (·····*·····) for various shapes.

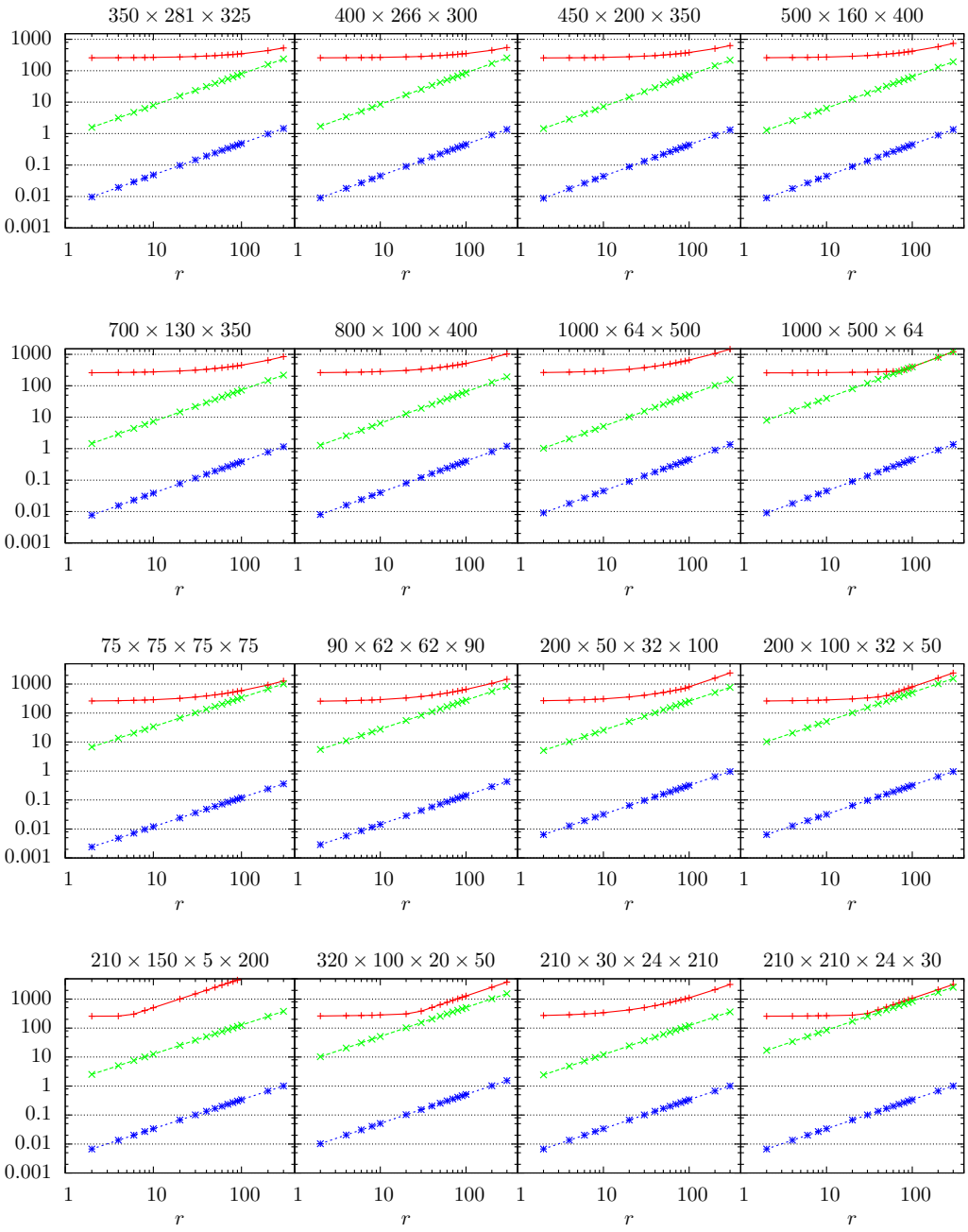

FIGURE 5. Additional memory requirements in Mb of the $*$-MTVP with r vectors for Ref. M1 (—+—), Ref. M3 (----×----), and Ref. M3B (·····*·····) for the shapes in Figure 4.

and in contrast to Ref. M1. Similarly the performance of a 1-MTVP is good if n_d is sufficiently large. Consequently, as soon as two factors are large, it is possible to rearrange the factors of the tensor once prior to all computations so that the effective throughput will be near optimal for all $k = 1, \ldots, d$. If only one factor is large, a rearrangement of the factors can be chosen so that the performance will be near optimal for every k-MTVP with $k \neq 1$. Conversely, the performance of Ref. M1 may be expected to be good only for the large factors.

5. Conclusions and future work

In this work we expounded upon successive contractions, a technique which we believe has not been disseminated widely. We have shown that it is effective at eliminating memory operations that would otherwise be required for the construction of explicit unfoldings; for k-TVPs and k-MTVPs with a small number of vectors the cost of computing unfoldings was demonstrated to severely impact performance, both in terms of time and space. We illustrated that current techniques for computing k-MTVPs are prohibitively expensive in terms of memory consumption whenever the number of vectors to multiply with is large relative to the size of the factors of the tensor. For resolving this issue we explored blocking techniques for tensors and derived the correct blocked equivalent of the k-MTVP, which we believe to be a novel idea. By subdividing the tensor into subtensors of suitable size, the additional memory requirements can be restricted to a maximum deemed tolerable by the user. We advocated in particular the use of slicing in the first two factors, so that explicit subtensor storage is not required for obtaining good performance while enabling the use of well-established an optimized matrix libraries. In addition, no additional code would need to be written for performing operations on tensors stored with subtensor storage.

With the proposed Ref. M3B implementation of the k-MTVP a reduction of the additional memory requirements is immediate, however, for attaining competitive execution times it is imperative that the sizes of the subtensors be appropriately chosen. We have argued that if the factors of the tensors have been reordered so that $n_1 \geq n_2 \geq \ldots \geq n_d$ then slicing in the first two factors often yields good performance. It was confirmed experimentally that the blocked implementation Ref. M3B typically outperforms the widely disseminated Ref. M1 approach in terms of execution time. Essentially Ref. M3B may be expected to attain appreciable performance gains whenever the matrix slices are chosen as large and as square as possible. It is imperative to realize that this ability to choose the shape of the required matrix product, within some restrictions imposed only by the shape of the tensor, is a key advantage of the proposed method with successive contractions over the classic approach. In the latter the matrix shape depends not only on the shape of the tensor, which is an immutable constraint, but also on the factor k in which a vector is generated by the k-MTVP. This immediately entails bad performance if n_k is small in absolute terms, say less than 200 with our Eigen-based implementation. Ref. M3B, on the other hand, will attain high throughput if two factors of the tensor are large, i.e., larger than about 200, in an absolute sense.

We believe that we have only scratched the surface regarding high-quality performance-oriented implementations of key operations on tensors, and hope that this work proves to be a stepping stone from which more thorough studies of the k-MTVP and related operations may spring. In particular, we mention that a good strategy for performing k-MTVPs with tensor shapes in which all factors are small is still missing. In this work we were only concerned with sequential algorithms for dense tensors. To our knowledge several other settings have not yet attracted sufficient interest, despite their important applications. We mention optimized data structures and corresponding algorithms for k-MTVPs with sparse tensors, out-of-core algorithms for huge dense tensors, and parallel implementations. Besides the k-MTVP, other important operations, such as the multilinear multiplication, also warrant further research.

Acknowledgements

We thank A.-J. Yzelman for interesting and engaging discussions. L. Sorber is thanked for making available to us the LaTeX package used to generate Figure 1.

References

E. Acar, D.M. Dunlavy, T.G. Kolda, and M. Mørup. 2011. Scalable tensor factorizations for incomplete data. *Chemometr. Intell. Lab.* 106, 1 (2011), 41–56.

A. Ammar, F. Chinesta, and A. Falcó. 2010. On the convergence of a greedy rank-one update algorithm for a class of linear systems. *Arch. Comput. Methods Eng.* 17 (2010), 473–486.

C.A. Andersson and R. Bro. 2000. The N-way Toolbox for MATLAB. *Chemometr. Intell. Lab.* 52, 1 (2000), 1–4.

B.W. Bader and T.G. Kolda. 2006. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Trans. Math. Software* 32, 4 (2006), 635–653.

B.W. Bader and T.G. Kolda. 2012. MATLAB Tensor Toolbox Version 2.5. (March 2012). `http://csmr.ca.sandia.gov/~tgkolda/TensorToolbox/`

R. Bro and C.A. Andersson. 1998. Improving the speed of multiway algorithms: Part II: Compression. *Chemometr. Intell. Lab.* 42, 1-2 (1998), 105–113.

J. Carroll and J.-J. Chang. 1970. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart–Young" decomposition. *Psychometrika* 35, 3 (1970), 283–319.

J. Chang, W. Sun, and Y. Chen. 2010. A modified Newton's method for best rank-one approximation to tensors. *Appl. Math. Comput.* 216 (2010), 1859–1867.

B. Chen, S. He, Z. Li, and S. Zhang. 2012. Maximal block improvement and polynomial optimization. *SIAM J. Optim.* 22, 1 (2012), 87–107.

Y. Chen. 2012. Successive unconstrained dual optimization method for rank-one approximation to tensors. *J. Appl. Math. Comput.* 38 (2012), 9–23.

L. De Lathauwer. 2008. Decompositions of a higher-order tensor in block terms—Part II: Definitions and uniqueness. *SIAM J. Matrix Anal. Appl.* 30, 3 (2008), 1033–1066.

L. De Lathauwer, B. De Moor, and J. Vandewalle. 2000a. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.* 21, 4 (2000), 1253–1278.

L. De Lathauwer, B. De Moor, and J. Vandewalle. 2000b. On the best rank-1 and rank-(R_1, R_2, ..., R_N) approximation of higher-order tensors. *SIAM J. Matrix Anal. Appl.* 21, 4 (2000), 1324–1342.

V. de Silva and L.-H. Lim. 2008. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM J. Matrix Anal. Appl.* 30, 3 (2008), 1084–1127.

J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software* 16, 1 (1990), 1–17.

J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Software* 14, 1 (1988), 1–17.

L. Eldén and B. Savas. 2009. A Newton–Grassmann method for computing the best multilinear rank-(r_1, r_2, r_3) approximation of a tensor. *SIAM J. Matrix Anal. Appl.* 31, 2 (2009), 248–271.

A. Falcó and A. Nouy. 2012. Proper generalized decomposition for nonlinear convex problems in tensor Banach spaces. *Numer. Math.* 121 (2012), 503–530.

S.A. Goreinov, I.V. Oseledets, and D.V. Savostyanov. 2012. Wedderburn rank reduction and Krylov subspace method for tensor approximation. Part 1: Tucker case. *SIAM J. Sci. Comput.* 34, 1 (2012), A1–A27.

K. Goto and R.A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Software* 34, 3 (2008), 1–24. Article 12.

G. Guennebaud, B. Jacob, and others. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

J.A. Gunnels, F.G. Gustavson, G.M. Henry, and R.A. van de Geijn. 2001. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Software* 27, 4 (2001), 422–455.

F.G. Gustavson. 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. of Res. Dev.* 41, 6 (1997), 737–755.

R. A. Harshman. 1970. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics* 16 (1970), 1–84.

C. Hayashi and F. Hayashi. 1982. A new algorithm to solve PARAFAC-model. *Behaviormetrika* 11 (1982), 49–60.

F.L. Hitchcock. 1927. Multiple invariants and generalized rank of a p-way matrix or tensor. *J. Math. Phys.* 7 (1927), 39–79.

K. Iglberger, G. Hager, J. Treibig, and U. Rüde. 2012. Expression templates revisited: a performance analysis of current methodologies. *SIAM J. Sci. Comput.* 34, 2 (2012), C42–C69.

Intel. 2013. Intel Math Kernel Library. http://software.intel.com/en-us/intel-mkl. (2013).

E. Kofidis and P.A. Regalia. 2002. On the best rank-1 approximation of higher-order supersymmetric tensors. *SIAM J. Matrix Anal. Appl.* 23, 3 (2002), 863–884.

T.G. Kolda and B.W. Bader. 2009. Tensor decompositions and applications. *SIAM Rev.* 51, 3 (2009), 455–500.

T.G. Kolda and J.R. Mayo. 2011. Shifted power method for computing tensor eigenpairs. *SIAM J. Matrix Anal. Appl.* 32, 4 (2011), 1095–1124.

P. Kroonenberg and J. de Leeuw. 1980. Principal component analysis of three-mode data by means of alternating least squares algorithms. *Psychometrika* 45, 1 (1980), 69–97.

L.S. Lorente, J.M. Vega, and A. Velazquez. 2010. Compression of aerodynamic databases using high-order singular value decomposition. *Aerosp. Sci. Technol.* 14, 3 (2010), 168–177.

G. Morton. 1966. *A computer oriented geodetic data base and a new technique in file sequencing.* Technical Report. IBM, Ottawa, Canada.

M. Mørup. 2011. Applications of tensor (multiway array) factorizations and decompositions in data mining. *WIREs Data Mining Knowl. Discov.* 1, 1 (2011), 24–40.

I.V. Oseledets and D.V. Savost'yanov. 2006. Minimization methods for approximating tensors and their comparison. *Comp. Math. Math. Phys+* 46, 10 (2006), 1641–1650.

P. Paatero. 1997. A weighted non-negative least squares algorithm for three-way 'PARAFAC' factor analysis. *Chemometr. Intell. Lab.* 38 (1997), 223–242.

A.-H. Phan, P. Tichavský, and A. Cichocki. 2013a. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Trans. Signal Process.* 61, 19 (2013), 4834–4846.

A.-H. Phan, P. Tichavský, and A. Cichocki. 2013b. Low complexity damped Gauss–Newton algorithms for CANDECOMP/PARAFAC. *SIAM J. Matrix Anal. Appl.* 34, 1 (2013), 126–147.

G. Quintana-Ortí, M. Igual, F.D. Marqués, E.S. Quintana-Ortí, and R.A. van de Geijn. 2012. A runtime system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures. *ACM Trans. Math. Software* 38, 4, Article 25 (2012), 25 pages.

G. Quintana-Ortí, E.S. Quintana-Ortí, R.A. van de Geijn, F.G. Van Zee, and E. Chan. 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Software* 36, 3, Article 14 (2009), 26 pages.

S. Ragnarsson and C.F. Van Loan. 2012. Block tensor unfoldings. *SIAM J. Matrix Anal. Appl.* 33, 1 (2012), 149–169.

B. Savas and L. Eldén. 2013. Krylov-type methods for tensor computations I. *Linear Algebra Appl.* 438, 2 (2013), 891–918.

M.D Schatz, T.M. Low, R.A. van de Geijn, and T.G. Kolda. 2013. Exploiting symmetry in tensors for high performance. (2013). arXiv:1301.7744.

L. Sorber, M. Van Barel, and L. De Lathauwer. 2013a. Optimization-based algorithms for tensor decompositions: canonical polyadic decomposition, decomposition in rank-$(L_r, L_r, 1)$ terms, and a new generalization. *SIAM J. Optim.* 23, 2 (2013), 695–720.

L. Sorber, M. van Barel, and L. de Lathauwer. 2013b. Tensorlab v1.0. (February 2013). `http://esat.kuleuven.be/sista/tensorlab/`

G. Tomasi and R. Bro. 2005. PARAFAC and missing values. *Chemometr. Intell. Lab.* 75 (2005), 163–180.

L.R. Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.

N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen. 2012. A new truncation strategy for the higher-order singular value decomposition. *SIAM J. Sci. Comput.* 34, 2 (2012), A1027–A1052.

Y. Wang and L. Qi. 2007. On the successive supersymmetric rank-1 decomposition of higher-order supersymmetric tensors. *Numer. Linear Algebra Appl.* 14, 6 (2007), 503–519.

R. C. Whaley and A. Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software Pract. Exper.* 35, 2 (2005), 101–121.

R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Comput.* 27, 1–2 (2001), 3–35.

T. Zhang and G.H. Golub. 2001. Rank-one approximation to high order tensors. *SIAM J. Matrix Anal. Appl.* 23, 2 (2001), 534–550.

Appendix A. Matlab implementation of Reference S3

```
function [A] = tvp(A,v,k,col)
  if nargin < 4; col = 1; end
  Asize = [size(A) ones(1,length(v)-ndims(A))];
  d = length(v);
  for l = 1 : k-1
      M = reshape(A, Asize(l), []);
      A = (M' * v{l}(:,col))';
      Asize(l) = 1;
  end
  for r = d : -1 : k+1
      M = reshape(A, [], Asize(r));
      A = M * v{r}(:,col);
      Asize(r) = 1;
  end
end
```

Appendix B. Matlab implementation of Reference M3

```
function [ C ] = M3( T, V, k )
  Tsize = size(T);
  C = zeros( Tsize(k), size(V{1},2) );
  d = length( Tsize );
  scal = ones(1, size(V{1},2));
  imath = 1;
  while (Tsize(imath)==1) && (imath < k)
      scal = scal .* V{imath};
      imath = imath + 1;
  end
  if k > imath
    T = reshape( T, Tsize(imath), [] );
    A = (T' * V{imath})';
    Tsize(imath) = 1;
    for r = 1 : size(A,1)
      B = reshape( A(r,:), Tsize(imath+1:end) );
      C(:,r) = tvp( B, V(imath+1:end), k-imath, r );
    end
  else
    scal = ones(size(scal));
    jmath = d;
    while (Tsize(jmath)==1) && (jmath > k)
        scal = scal .* V{jmath};
        jmath = jmath - 1;
    end
    T = reshape( T, [], Tsize(jmath) );
    A = T * V{jmath};
    Tsize(jmath) = 1;
    for c = 1 : size(A,2)
      B = reshape( A(:,c), Tsize );
      C(:,c) = tvp( B, V(1:jmath-1), k, c );
    end
  end
  C = C * diag(scal);
```

end

E-mail address: nick.vannieuwenhoven@cs.kuleuven.be

Department of Computer Science, KU Leuven, Belgium

E-mail address: nickvanbaelen@gmail.com

Materialise NV, Leuven, Belgium

E-mail address: karl.meerbergen@cs.kuleuven.be

Department of Computer Science, KU Leuven, Belgium

E-mail address: raf.vandebril@cs.kuleuven.be

Department of Computer Science, KU Leuven, Belgium